

David M. Dill



The

COBOL Presentation Manager Programming Guide



The COBOL

Presentation Manager Programming Guide

For OS/2 Versions 1.3 and 2.0

The COBOL

Presentation Manager Programming Guide

For OS/2 Versions 1.3 and 2.0

David Dill



VAN NOSTRAND REINHOLD
New York

Copyright © 1992 by Van Nostrand Reinhold

Library of Congress Catalog Card Number 92-10877
ISBN 0-442-01293-4

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without written permission of the publisher.

The right to copy, alter or make use of the sample code, as desired by the reader, is granted. Written permission is waived for use of the sample code.

Manufactured in the United States of America

Van Nostrand Reinhold
115 Fifth Avenue
New York, New York 10003

Chapman and Hall
2-6 Boundary Row
London, SE1 8HN, England

Thomas Nelson Australia
102 Dodds Street
South Melbourne 3205
Victoria, Australia

Nelson Canada
1120 Birchmount Road
Scarborough, Ontario M1K 5G4, Canada

16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Library Of Congress Cataloging-In-Publication Data
Dill, David M., 1941-

The COBOL presentation manager programming guide / David M. Dill. -

- 1st ed.

p. cm.

Includes index.

ISBN 0-442-01293-4

1. COBOL (computer program language) 2. Presentation manager
(Computer program) I. Title.

QA76.73.C25D56 1992

005.13'3—dc20

92-10877

CIP

Trademarks

Animator, Xilerator, Micro Focus COBOL/2 and Micro Focus Toolkit are trademarks or registered trademarks of Micro Focus Limited.

IBM ®, OS/2 ® and PS/2 ® are registered trademarks of International Business Machines Corporation.

Intel and 8088, 80286, 80386 and 80486 are trademarks of Intel Corporation.

ITC Avant Garde Gothic ® is a registered trademark of International Type Face Corporation Of America.

Microsoft ® is a registered trademark of Microsoft Corporation

New Century Schoolbook is a trademark of Linotype-Hell AG.

Operating System/2, Presentation Manager and System Application Architecture are trademarks of International Business Machines Corporation.

PostScript is a trademark of Adobe Systems Incorporated.

Times Roman is a trademark of Linotype AG.

Windows is a trademark of Microsoft Corporation.

Preface

Growing support for OS/2 and its Presentation Manager (PM) graphical interface has created a problem for many companies. With hundreds of COBOL applications, millions of lines of COBOL source code and legions of experienced COBOL programmers, companies are now being told that to use OS/2 and the Presentation Manager effectively, they must adopt the C Language as their programming language, retrain their programmers and rewrite their existing applications.

This understandably creates concern. As they wrestle with limited data processing resources, companies cannot always justify rewriting stable, well-performing applications for the sake of a new interface or a new platform.

The perplexing part of this situation is that most companies agree that OS/2 is the best platform for future growth and many indicate that OS/2 will eventually be their personal computer operating system of choice. The problem is how to get from here to there at a reasonable cost and within a reasonable time period.

Companies have options when considering OS/2 and PM as a potential programming environment and they should carefully examine each option. There are alternatives to the expensive retraining and rewriting necessary with a simultaneous conversion to the C language and the Presentation Manager.

Contrary to popular myth, C is not the only language supported by the OS/2 Presentation Manager environment. COBOL may also be used to produce effective PM applications. While COBOL may not be correct for every task and designers and programmers may still need some C programming experience, for the vast majority of a company's PC applications, COBOL is a real and viable alternative to writing in C.

The ability to use COBOL as the primary programming language within the PM environment can drastically alter the cost equation for an OS/2 PM conversion.

Companies that cannot justify the move based upon a simultaneous conversion to the C Language and PM should take another look at the cost and time involved with the C language conversion removed from the equation.

This book is designed to help COBOL programmers understand the COBOL - Presentation Manager programming environment and to help them make a more informed decision about using COBOL with the Presentation Manager. Additionally, this book should be the starting point for COBOL programmers to begin their Presentation Manager programming training.

Contents

Preface	vii
Introduction	1
What I Assumed You Know About PM And COBOL	2
How To Use This Book	3
A Word About The Sample Program	3
Software And Documentation That You Will Need	4
Chapter 1. The OS/2 And Presentation Manager Environment	7
The OS/2 Operating System	8
The Presentation Manager Environment	13
A Word About Presentation Manager Windows	14
A Word About Window Classes	17
Establishing A Window's Look And Feel	18
The Window Hierarchy	21
The Message-Based PM System	23
The Flow Of Information Within The Presentation Manager	25
OS/2 Version 2.0	27
The Flat Memory Model	28
The Process Address Space	29
Memory Management	30
Page Management	31
The Guard Page	33
The Memory Management Process	33
OS/2 16-Bit Memory Compatibility	33
Thunk Layers	34
The Workplace Shell	35
Compatibility With Version 1.3 Programs	36
Running Current Compilers Under Version 2.0	38

x The COBOL Presentation Manager Programming Guide

Migrating Existing Programs To Version 2.0	38
Chapter 2. Using COBOL With The Presentation Manager	45
Programming Window Procedures	46
An Extended Compiler vs. The COBOL/2 Bindings	48
Using The C Bindings	49
Implementing The Pascal Calling Convention	50
Call Parameter Passing	51
The Use Of Null-Terminated Strings	52
Working With Intel Reverse Order storage Values	54
Using The Returning Parameter	55
Initializing Numeric Variables	55
Static Linked PM Calls	55
Working With Handles	56
Some Coding Conventions You Should Use	57
The Essential Parts Of A COBOL PM Program	57
The Initialization Routine	58
The Message Processing Routine	58
Window Procedures	60
The Termination Routine	60
Chapter 3. Getting Started	61
Setting Up Your Development Environment	61
Modifying Your CONFIG.SYS File	62
The Compile And Link Command File	63
COBOL Compiler Directives	65
The Linker Response File	66
The Module Definition File	68
The Resource File	70
The Resource Compiler Command File	72
The Icon Editor	73
The Dialog Box Editor	74
The Font Editor	75
Chapter 4. Coding The Basic PM Program	77
A Word About The Sample Program	77
The Identification Division	79
The Environment Division	79
The Working-Storage Section	80
The Local-Storage Section	83
The Linkage Section	84
The Procedure Division	85

Coding The WinInitialize Call	86
Coding The WinCreateMsgQueue Call	86
Coding The WinRegisterClass Call	87
Coding The WinCreateStdWindow Call	88
Sizing And Positioning The Window	90
Measuring The Size Of The Desktop	90
Positioning The Window	91
Building The Main Message Processing Routine	94
Coding The WinGetMsg Call	94
Coding The WinDispatchMsg Call	96
Ending The Presentation Manager Program	97
Coding The WinDestroyWindow Call	98
Coding The WinDestroyMsgQueue Call	98
Coding The WinTerminate Call	98
The Window Procedure	99
Coding The WinDefWindowProc Call	100
The Basic COBOL PM Program	102
 Chapter 5. Working With Resources	109
A Word About Resources	109
Coding Conventions Unique To The Resource Script File	111
Creating The Resource Script File	111
Defining The Menu Entry	114
Creating The Resource Header File	116
Compiling The Resource Script File	117
Modifying The COBOL Program	117
 Chapter 6. Processing PM Messages	121
A Word About Messages	121
Message Recursion	124
How To Check For Selected Messages	125
Processing Messages For A Window	126
A Word About Presentation Spaces	128
Coding The WinBeginPaint Call	129
Coding The WinFillRect Call	130
Coding The WinEndPaint Call	132
 Chapter 7. Beginning The User Dialog Process	133
The Message Box	134
Adding Pull-Down Menus	136
Modifying The Resource Script File	139
Accelerator Keys	140

Adding The Accelerator Key Table	141
Adding The Resource File String Table	143
Accessing String Data In The Resource File	144
Coding The WinLoadString Call	145
Coding The WinMessageBox Call	146
Processing The WM-COMMAND Message	147
Coding The Command Routine	148
 Chapter 8. Icons And Mouse Pointers	151
Creating An Icon Or Mouse Pointer	152
Adding The Icon To The Resource Script File	155
Creating A Custom Mouse Pointer	157
Adding The Mouse Pointer To The Resource Script File	158
Adding The Pointer Code To The Sample Program	159
Coding The WinLoadPointer Call	159
Checking For The Movement Of The Mouse	161
Coding The WinQueryFocus Call	161
Coding The WinSetPointer Call	162
 Chapter 9. Creating And Sending PM Messages	163
Control Messages Within The Sample Program	165
Obtaining The Handle Of The Target Control Window	165
Defining The Message Parameters	167
Coding The WinSendMsg Call	169
Updating The Resource Script File	170
Adding The Resource File Changes To The Sample Program	172
Registering The Child Window	172
Intercepting The MI-Source Message	173
Destroying Child Windows	174
 Chapter 10. Working With Dialogs	177
Modal vs. Modeless Dialog Boxes	177
The Program Termination Dialog Box	178
Creating The Dialog Box Template	180
Using The Dialog Editor	180
Interpreting The Dialog Template	182
The Generated Dialog Template Format	186
Updating The Resource Script File And Application	187
Generating The WM-QUIT Message internally	189
Coding The WinPostMsg Call	189
Coding The WinDlgBox Call	190
The Dialog Process Message Flow	192

Coding The Dialog Procedure	193
Coding The WinDefDlgProc Call	194
Ending The Dialog	195
Coding The WinDismissDlg Call	195
Interpreting The Dialog Results	196
 Chapter 11. More About Dialogs	199
The Types Of Dialog Control Windows	200
The Notification Message	202
Control Messages	202
Passing And Receiving Dialog Data	203
Passing The Data Structure Pointer	204
Establishing Structure Addressability	205
The PM Call Dialog	206
Creating The PM Call Dialog Template	207
Updating The Resource Script File	207
Updating The Program's Working Storage Section	208
Building The PM Call Dialog Box	210
The Dialog Procedure Initialization Routine	212
Loading The List Box	214
Processing The Dialog	216
Ending The Dialog	218
 Chapter 12. Working With Text And Fonts	221
A Word About Fonts	221
Image vs. Outline Fonts	223
Selecting The Desired Font	226
Establishing The Current Font	228
Testing For The Required Font	231
Retrieving The Data To Be Displayed	231
Modifications To The Sample Program	233
Coding The GpiQueryFonts Call	234
Parsing The Font Metrics File	234
Coding The GpiCreateLogFont Call	238
Coding The GpiSetCharSet Call	238
Coding The GpiQueryFontMetrics Call	239
Coding The GpiCharStringAt Call	241
Resetting The Current Font	243
Coding The GpiDeleteSetId Call	243
Changing The Look Of Other Windows	244
Coding The WinSetPresParam Call	246
 Chapter 13. Buttons, Boxes And Entry Fields	247

Radio Buttons	247
Check Boxes	248
Entry Fields	250
Grouping User Controls	251
Creating The Window Edit Dialog Box	253
Updating The Resource Script File	253
Adding The MI-Wedit Command Routine	254
Dialog Initialization Processing	255
Setting The Default Dialog Controls	255
Coding The Window Edit Dialog Procedure	258
Processing Check Box Data	258
Processing Radio Button Data	260
Processing Entry Field Text	260
Processing The Dialog's Returned Data	263
Window Enumeration	265
Coding The WinBeginEnumWindows Call	267
Coding The WinGetNextWindow Call	268
Coding The WinEndEnumWindows Call	268
Changing The Color Of Window Text	268
Chapter 14. System And Application Profiles	271
The System Profiles	272
The Structure Of Profiles	273
Application Profiles	278
Data Types Used In Profiles	280
Processing User Profiles	281
Coding The PrfOpenProfile And PrfClose Profile Calls	281
The Sample Program's Startup Processing	282
Coding The WinUpper Call	282
Coding The PrfQueryProfileString Call	283
Querying For The Sample Program's Customization Strings	284
Coding The PrfQueryProfileInt Call	286
Writing To A Profile	287
Coding The PrfWriteProfileString Call	288
Coding The PrfWriteProfileData Call	288
Chapter 15. Window Subclassing	291
Implementing Window Subclassing	292
Subclassing A Single Window Or An Entire Class	293
A Word About Window Words	294
Creating The Frame-Grabbing Pointers	297
Making Changes To The Sample Program	297
Coding The WinSubclassWindow Call	298

Loading Window Words	300
The Subclassing Window Procedure	300
Coding The WinQueryFocus Call	301
Coding The WinQueryWindowULong Call	301
Sending Messages On To The Original Procedure	302
Chapter 16. Dynamic Linking	305
Public And Private Dynamic Link Libraries	306
Preload And Loadoncall Dynamic Link Libraries	308
Creating The Linker Response File	308
Creating The Module Definition File	309
Building An Import Library	313
Building A Dynamic Link Library	314
Resource-Only Dynamic Link Libraries	315
The Resource Stub Program	315
The Printer Setup Dialog Procedure	318
Coding The DosLoadModule Call	319
Coding The DosFreeModule Call	320
Modifying The Sample Program	322
Chapter 17. Printing	323
The OS/2 Printer Subsystem	324
The User Interface Dialogs	324
The Queue Processor	324
The Printer Device Drivers	325
The Kernel Device Drivers	325
Data Flow	326
Basic Printing	326
Queued Printing	326
Direct Printing	327
Printer And Job Properties	328
The Printer Data Stream	330
Raw Data Streams	330
Standard Data Streams	330
Building A Standard Data Stream	331
Building A Raw Data Stream	336
Appendix A. The Sample Program	343
Appendix B. PM COBOL Function Calls	415
Appendix C. PM Messages	427
Appendix D. OS/2 Version 2.0 PM Header Definitions	429
Appendix E. Presentation Manager And Kernel Calls	461

The COBOL

Presentation Manager Programming Guide

For OS/2 Versions 1.3 and 2.0

Introduction

It has been said that the hardest part of dieting, exercising or any long-running task is just getting started. The same can be said for programming, especially when programming within the OS/2 Presentation Manager environment. For those who are not familiar with the C Programming Language, there seems to be an entirely new coding structure associated with the Presentation Manager. So many new rules and new calls are involved that just getting started often seems too much of a hassle to make the effort. If only there were a place where programmers could get a head start, a kind of fast path to the first window, it would make getting started much easier.

This book is intended to be such a place. It is not designed to be the definitive resource on Presentation Manager programming, but rather, an introduction to PM written especially for those programmers who consider COBOL to be their primary programming language. The C Language is not spoken here. This book assumes no prior knowledge of OS/2, the Presentation Manager or the C programming language, but it does assume a basic knowledge of COBOL. For COBOL programmers, this book will provide the following.

- An introduction to the OS/2 and PM environment
- An introduction to programming PM using COBOL
- A COBOL coding reference for the most common PM calls
- The COBOL code that forms the nucleus of every PM Program

This book is designed to work with all versions of OS/2 that support the Presentation Manager, including the just released Version 2.0. Initial development was done using Version 1.3, and in order to make this book more useful to all COBOL programmers, appropriate information on architecture, limitations and restrictions of both Version 1.3

2 The COBOL Presentation Manager Programming Guide

and Version 2.0 are included. Some of the PM and OS/2 calls have changed during their conversion to the 32-bit architecture of OS/2 Release 2.0 and a complete appendix showing these changes is included. Examples for both Version 1.3 and Version 2.0 are shown and the sample program runs on both versions.

What I Assumed You Know About PM And COBOL

In writing this book I assumed no special knowledge of the Operating System/2 or the Presentation Manager, it's graphical user interface. While I have attempted to explain every line of Presentation Manager code and every PM call within the sample program, there is no requirement that you understand every detail of a PM call to use the code. There are certain PM calls and certain coding routines that are required in every PM program and you can simply cut and paste my code to form the basis of your programs. The sample program shown at the end of Chapter 4 forms the nucleus of every COBOL PM program that you write. As a result, I have separated this code from the full sample program shown in Appendix A, so that you can use it, as is, to build your own COBOL PM programs.

The Presentation Manager environment is a totally unique personal computer operating environment, and writing programs for this environment is different from any other COBOL PC programming effort you have ever undertaken. This book will give you a beneficial general understanding of the Presentation Manager and the requirements it places on programs that must be followed when writing a PM program. Understanding how messages flow within the Presentation Manager system and how a program communicates with PM will make it easier to understand the calls and why they work the way they do. If you have no understanding of the Presentation Manager, some background reading beyond what is included within this book is suggested.

Similarly, I have made no specific assumptions about your level of COBOL knowledge. While some COBOL experience is assumed, I kept the code very basic for two reasons. First, keeping the COBOL code simple makes it easier to see the PM-related calls and understand how they fit together within a COBOL program structure. Second, I assumed that my sample programs would be used as the basic structure for your programs and that you would cut and paste my basic PM structure into your program, adding the additional code that you require. Fitting in your code is easier when there is less of my code.

When having to choose between what may appear to be poor or inefficient coding versus better, more compact coding for a particular COBOL function, I sometimes chose the less efficient code if it made the particular PM function being discussed easier to understand. I expect that experienced COBOL programmers will have better ways to code the

COBOL routines, and you should make whatever changes you feel are appropriate. My objective, after all, is to teach the PM interface for COBOL programs.

How To Use This Book

Most PM programming books tend to immerse the reader in individual PM topics, covering a specific topic entirely before moving on to another, perhaps unrelated topic. This complex arrangement of chapters together with the depth of discussion in each chapter can make it difficult for beginners to understand how everything fits together. It is often difficult to see the PM forest for all of the application programming interface (API) trees.

This book is constructed differently. Its designed to be read from the beginning to the end, to be used as an education tool rather than a reference source. It groups the PM calls as they relate to building a part of a window or dialog, not by an artificial PM collection, such as all Gpi calls. For example, you will find a discussion of Resource Script files in a chapter about adding menus to the frame window, because Resource Script files are required when working with menus. It is hoped that this slow progression through the PM complexities will help the reader to a better understanding how all the pieces fit together. This structure does not make the best reference manual, but an extensive index has been provided to make it easier to find specific points within the book.

A Word About The Sample Program

The sample program was designed to highlight the use of particular Presentation Manager calls and explain particular Presentation Manager functions and not to be a shining example of a PM program. Thus, while the sample program does not perform any useful functions, it does contain all of the basic PM routines and many of the most common PM calls that you'll need when you write your PM programs.

The sample program was written using the Micro Focus COBOL/2 compiler, and many of the COBOL terms and phrases utilized - everything from the calling convention implementation to the COMP-5 phrase to the expanded Procedure headers - are exclusive to the Micro Focus compiler. Where the Micro Focus compiler offered an extension that worked well with PM, I used it. If you are not using the Micro Focus compiler, you will need to change these Micro Focus implementations to match the compiler that you are using.

4 The COBOL Presentation Manager Programming Guide

The complete source listing for the sample program, together with all of the header files, dialog procedures and dialog templates, is included in Appendix A. The sample program code is copyrighted, but you are encouraged to copy, modify and make use of any or all of the sample code.

As stated earlier, a source listing of the COBOL PM starter program, is included at the end of Chapter 4. This program can be used as the basis of every COBOL PM program that you write and it too may be copied and used as you see fit. *This code is your fast path to the first window.*

Software And Documentation That You Will Need

You will need to use several software products to build COBOL - Presentation Manager programs. They are listed below. The list includes only the major software components needed. Other items, such as an editor, are left to your discretion. The major software products you'll need to use with this book are:

- OS/2 Standard Edition or Extended Edition Version 1.3
 - or
 - OS/2 Version 2.0.
- The OS/2 Programming Tools and Information Kit Version 1.3
 - or
 - The Developer's Toolkit for Version 2.0.
- The Micro Focus COBOL/2 compiler or other COBOL compiler capable of producing OS/2 protect mode, reentrant programs that support external calls using the Pascal calling convention and store data using the Intel reverse order storage format. I recommend the Micro Focus COBOL/2 compiler and the associated COBOL/2 Toolset, but any COBOL compiler that meets the above criteria should work.
- A source debugger program. When programming for the PM environment, a source debugger is essential. Not only for the obvious debugging of your programs, but as a valuable learning tool as well. Much of what I learned about COBOL PM programming came from trial and error coding followed by long sessions with the Micro Focus Xilerator source debugger.

While no documentation is absolutely required for use with this book, some should be considered as a requirement when you launch into your COBOL PM programming. Most of the available documentation was written for use with the C Language and their

examples are C examples. While this limits the use of the examples, it does not diminish the value of the text that explains each call. Extracting information about particular PM calls from reference manuals written for C programmers, then translating the information into COBOL is a fact of life that we COBOL programmers will need to deal with for some time.

In the following list, I have included the Micro Focus COBOL/2 compiler manuals that I found of value. These manuals come with the Micro Focus COBOL/2 compiler and COBOL/2 Toolset. If you use another COBOL compiler, you should substitute the documentation that comes with your compiler. Here are some of the more important books I think you should keep handy.

OS/2 Version 1.3 Documentation

- Presentation Manager Programming Reference Volume 1 91F9261
- Presentation Manager Programming Reference Volume 2 91F9262
- Control Program Programming Reference 91F9260
- Programming Guide 91F9259
- Building Programs 91F9264
- Presentation Manager C/2 Bindings Reference 91F9265
- Programming Overview 91F9258

OS/2 Version 2.0 Documentation

- Presentation Manager Programming Reference Volume I S10G6264
- Presentation Manager Programming Reference Volume II S10G6265
- Presentation Manager Programming Reference Volume III S10G6272
- Control Program Programming Reference S10G6263
- Programming Guide Volume I GG243422
- Programming Guide Volume II G3620005
- Programming Guide Volume III SC282700
- SAA Common User Access Advanced 91 Guides SC344289

Micro Focus Documentation

- COBOL/2 Operating Guide 274-4339-641
- COBOL/2 Toolset PC Programmer's Guide 274-4345-641
- COBOL/2 Language Reference Volume 1 274-4353-641
- COBOL/2 Language Reference Volume 2 274-4371-641

Chapter 1

The OS/2 And Presentation Manager Environment

Since the introduction of the first IBM Personal Computer in the fall of 1981, hardware and software have been growing more complex and sophisticated. Personal computer hardware has led this surge of power and capacity. Today's 50-Mhz 80486 personal computers bear little resemblance to the original PC. Application software, too, has forged ahead, growing more complex and sophisticated, able to do the type of work that only a few years ago required large mainframe computers. During this period of rapid hardware and software growth, however, the PC-DOS operating system evolved little. After more than ten years of use it still retains many of its original design limitations.

The original PC-DOS was designed to support the Intel 8088/8086 family of microprocessors. The hardware capabilities of these early Intel microprocessors, referred to generically as the 808X microprocessors, influenced much of the design of PC-DOS. Many limitations in today's PC-DOS are a direct result of the design of the 8088/8086 microprocessors.

For example, the size of the 808X microprocessor address registers limited all personal computers based upon these microprocessors to 1 million bytes of memory. This memory size limitation led directly to the PC-DOS addressability design limitation of 1 million bytes. While this address space must have seemed more than adequate to the original designers, subtract from this limit the 360KB that the original PC carved out for hardware BIOS, and you have a very restrictive 640KB of memory available for today's complex programs. Reduce this further by the size of PC-DOS, LAN support programs, terminate-and-stay-resident programs and the usual assortment of device drivers, and the memory available for applications can hover just above 500KB.

The 808X microprocessors implemented a memory-addressing scheme called real-mode memory addressing. Real-mode addressing refers to the absolute relationship between a program's calculated memory address and the actual memory location where the data or instruction resides. That is to say, real-mode programs have the exact memory

8 The COBOL Presentation Manager Programming Guide

address where they expect to find the next instruction established at the time they are compiled and linked. Because of this method of memory addressing, programs running in real-mode own the memory they occupy and cannot be moved, segmented or manipulated by the operating system. PC-DOS uses this real-mode static memory management scheme.

The 808X microprocessors offer no memory protection and no protection of the hardware devices installed on the personal computer. Any program can address anywhere within the 1MB address space and effectively alter any available hardware device. Without microprocessor support for memory protection, the PC-DOS designers had no way to protect one program from another or PC-DOS from attack by a program. The lack of memory protection in PC-DOS is perhaps the biggest problem faced by programmers today when trying to implement bigger, more complex applications.

Much work has been done to try and overcome these major limitations that still plague today's PC-DOS systems. Most, however, are merely work-arounds and not solutions. Most do not offer the capabilities that today's complex multitasking environments demand. As a result, IBM and Microsoft began development of a new operating system, called Operating System/2, based upon the advanced hardware capabilities of the Intel 80286 microprocessor rather than the 808X.

It is not that OS/2 developers were that much smarter than the DOS developers, they weren't. But, what they did have was several years of personal computer usage and software development behind them, to better understand operating system requirements. More importantly, the advanced Intel 80286 and 80386 microprocessors were entering wide-spread use and were now a viable platform upon which to base an operating system.

The OS/2 Operating System

The resulting OS/2 system is a single-user operating system supporting a true multitasking environment with Protect-Mode virtual memory addressing and improved memory management. OS/2 Version 1.3 supports up to 16 million bytes of virtual memory per address space while Version 2.0, based upon the more advanced Intel 80386 microprocessor, supports an astounding 4 billion bytes of virtual address space per process.

To satisfy the memory requirements of today's applications, OS/2 implements a memory scheme that gives each program a large virtual address space, regardless of the actual memory installed on the computer. Through the use of a segmented memory and memory swapping, OS/2 manages the installed memory, moving parts of programs around in memory and swapping unused parts of programs to disk, to ensure that each program gets sufficient real memory to continue executing. Each program sees its

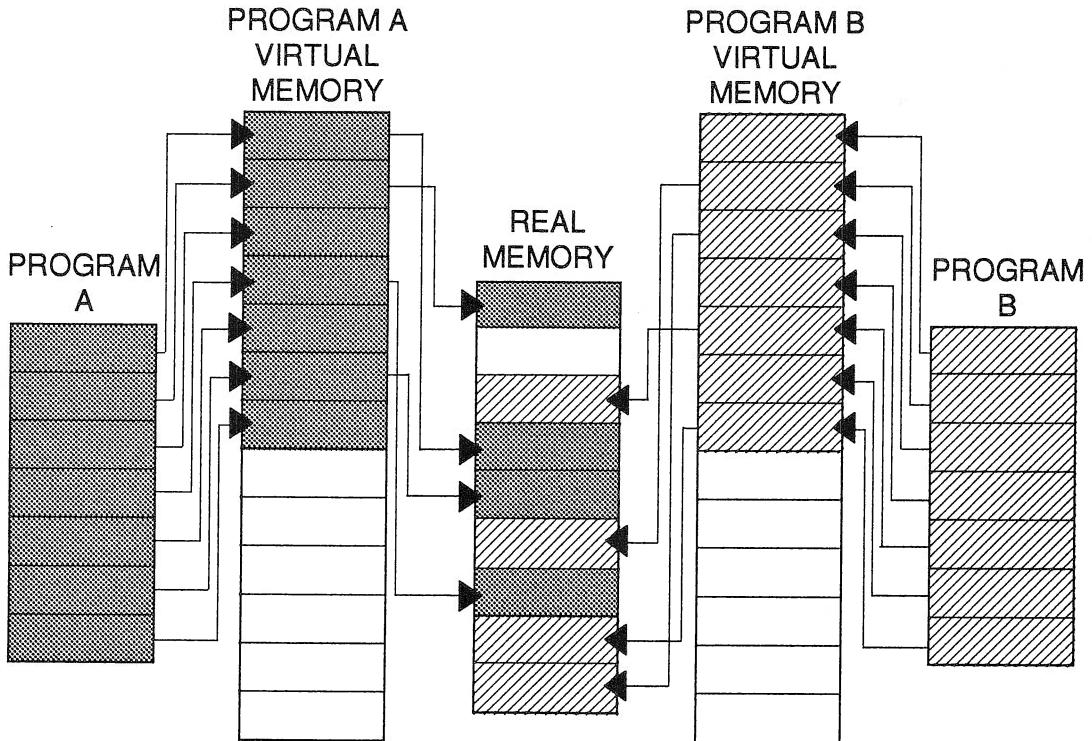


Figure 1-1 OS/2 virtual storage to real storage mapping

virtual memory as the complete personal computer memory and is unaware of the memory management performed by OS/2.

To support multiple concurrent program execution, OS/2 utilizes the Protected Virtual Addressing Mode of the 80286 or 80386 to provide a separate 16MB virtual address space for each Version 1.3 program and a 4GB virtual address space for each Version 2.0 program. OS/2 hides the amount and location of real memory from each program, giving each a virtual address space that it uses as its working memory. OS/2 uses the hardware protected mode addressing of the Intel microprocessor to confine each program to its own memory areas. Any attempt to address outside of its own memory results in the program being terminated without impact to the other programs running in the system. As a result, OS/2 is able to guarantee its integrity as well as that of each application.

Protect mode addressing does not map directly to real memory as with PC-DOS. Rather, the program's address registers point not to a real memory locations but to table entries that, in turn, point to a real memory locations. By manipulating the real memory pointers within these tables, OS/2 is free to move parts of a program around in memory or swap them to disk as required to best support all active applications.

10 The COBOL Presentation Manager Programming Guide

OS/2 Version 1.3 exploits the virtual segmentation hardware of the 80286 to offer segmented memory management. By managing each program as a series of variable-sized segments, up to 64KB in size, OS/2 needs to maintain only the most recently active segments of each program in memory. Then, when a segment is required that is not in memory, OS/2 swaps that segment into an available memory location and updates the segment's table entry to reflect the new memory location. As a result, a single application or a combination of applications larger than real memory may execute as if all the required real memory were available.

OS/2 Version 2.0 performs the same type of memory segmentation as Version 1.3, but uses fixed 4KB pages as the unit of segmentation rather than the variable 64KB segments used by Version 1.3. This allows for faster and more efficient memory management under Version 2.0.

Using the I/O instruction privilege levels of the Intel microprocessor (often referred to as ring levels), OS/2 is able to ensure the integrity of the installed hardware devices. Under OS/2, access to hardware devices is limited to code running at low privileged levels. Since applications run only at high privileged levels, their only access to hardware devices is via system calls to OS/2. This not only ensures that one program will not impact another's use of a hardware device, but allows multiple programs to access the same hardware simultaneously.

OS/2's primary unit of application control is the session. A session is allocated for each application with the PM running in one session. The session is a routing mechanism for user keyboard and mouse input, and application screen output. The session that is currently in control of the keyboard, mouse and screen is called the foreground session. The foreground session owns these physical devices, receiving all user input and controlling what is displayed on the screen. All other sessions are referred to as background sessions. Background sessions continue to execute, but do not receive user input and cannot display output. To receive input or display output, a background session must first be made the foreground session. OS/2 ensures that the input and output are routed to and from the foreground session.

Within a session, what we are accustomed to calling a program is referred to as a process. A process is the executing code and all of the resources associated with that execution at a point in time. A process, in turn, contains one or more units of execution, called threads. A thread is the unit of dispatch used by OS/2 to allocate processor execution cycles. All threads within a process share system resources equally, unless altered by the program (see Figure 1-2).

Multitasking allows multiple applications to run concurrently, each appearing to have a dedicated CPU and all required system resources. To support multitasking, OS/2 uses a round-robin time-slicing scheduler with preemptive dispatching to allocate specific amounts of processor time to each thread. To allow individual threads with critical response time requirements additional processor resources, OS/2 adds the concept of

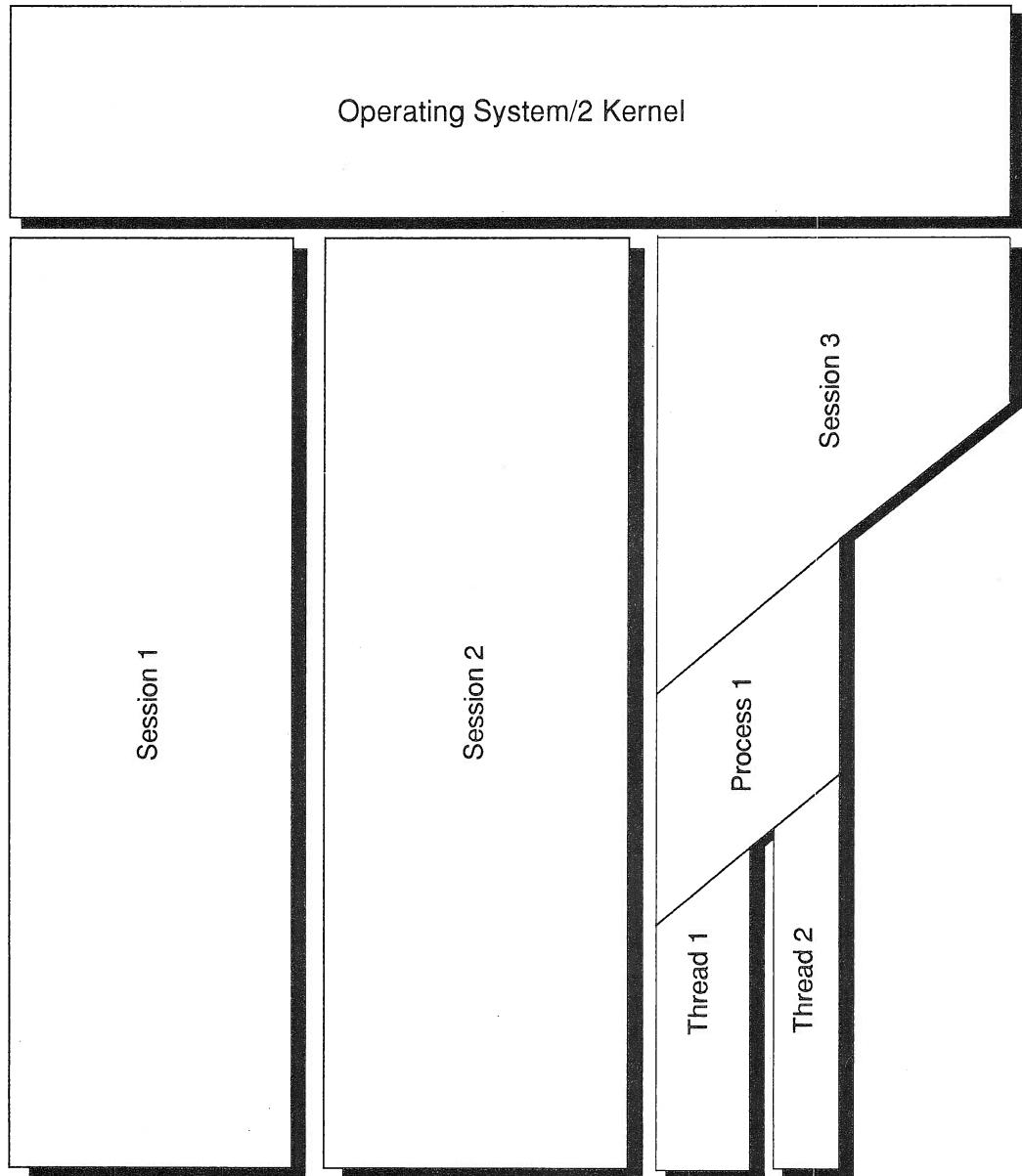


Figure 1-2 Organization of the OS/2 environment

12 The COBOL Presentation Manager Programming Guide

priority classes. Thread priority is used by OS/2 to determine which thread to dispatch next. On each interrupt, the OS/2 Dispatcher runs through the list of currently executing threads, and dispatches the thread with the highest priority that is ready to execute.

There are four classes of thread priority. Time-critical class has the highest dispatch priority, followed by the fixed high class, then the regular class, the normal dispatch priority and, finally, the idle class, which will be dispatched only when there is no higher class ready to execute. Within each class there are 32 priority levels that allow threads within a class to have priority over other threads within the same class. OS/2 uses these priority levels to automatically boost the priority of threads that are not receiving adequate system resources.

Unlike the PC-DOS interrupt-based service requests, OS/2 uses a call-return interface, with the hardware stack being used to pass the call parameters. In the multiple protection ring environment of the Intel microprocessor, significant performance gains are achieved by using this interface. The call's parameters are pushed onto the stack prior to the call, then the hardware copies these parameters from the requestor's stack to the receiving program's stack, thus giving optimum addressability and protection at minimal execution cost.

For COBOL programs, this call-return method of requesting service presents a major challenge. OS/2 requires call parameters be structured according to C Language specifications, in a manner referred to as the Pascal calling convention. COBOL programs use a reverse method of parameter passing (see Figure 1-3), thus making normal COBOL programs incapable of calling OS/2. Any COBOL compiler capable of producing protect-mode programs takes care of this problem for calls generated by the compiler, such as file I/O, but where the calls are not generated automatically by the compiler as with PM calls, the compiler must offer a way for the programmer to specify the Pascal calling convention.

The Intel protected mode call architecture offers additional benefits in the area of a program's structure. Application programs need only have their most commonly used routines loaded when they are started. Less commonly used routines may be left out of the initial program load, then loaded and linked dynamically only when they are called. This delayed linking, called dynamic linking, can significantly reduce a program's memory requirements. Since all OS/2 programs can access dynamically linked modules, they can share common routines and data contained in independent modules by automatically linking to these modules at the time the routines are called.

To allow separate and protected processes to communicate and pass data, OS/2 has a rich set of interprocess communication techniques. The three major classes of interprocess communications are shared memory, pipes and queues.

Shared Memory refers to memory made available to more than one process, allowing

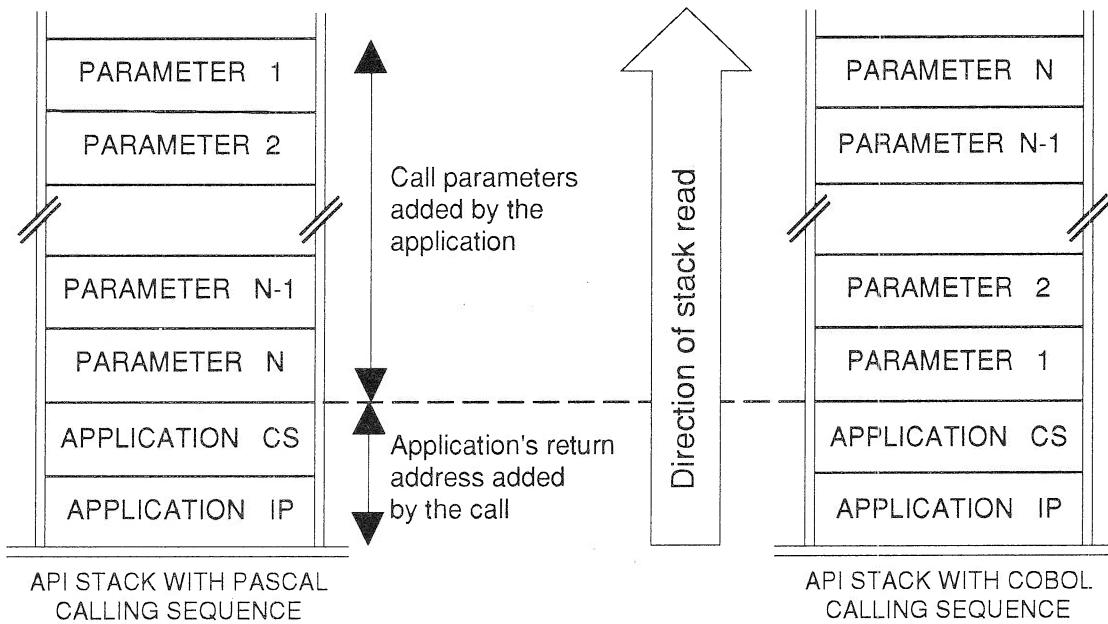


Figure 1-3 Format of the application's stack

those processes to read and/or write into that common memory area. Reading and writing to these common areas is usually controlled through the use of flags called semaphores.

Pipes allow processes to make data immediately available to other processes without the need to control access to shared memory. Data is passed from one process to another through a pipe function that resembles an in-core disk file. Data must be read on a first-in, first-out basis and once read, the data is destroyed.

Queues are like pipes, but no data is actually moved. Queues involve the use of pointers to gain access to data in a shared memory segment. Queues allow the reading process more flexibility in how the data is read, and when read, the data is not destroyed.

The Presentation Manager Environment

As was noted earlier, sessions are the highest level of multitasking control within the OS/2 environment. Each session can run one or more processes and OS/2 can run several sessions concurrently, presenting the user with the potential for managing a complex set of executing processes.

While sessions offer an excellent way for OS/2 to control concurrently executing processes, they are particularly poor as a method for the user to manage multiple processes. Sessions limit user interaction to the foreground session, preventing the user from viewing and interacting with all background processes. Sessions offer no assistance to the user in managing the OS/2 environment. Session control uses OS/2's text-based command line interface, that, like PC-DOS require the user to master a significant number of commands to effectively manage the sessions.

To overcome the drawbacks of managing at the session level, the Presentation Manager, with its graphical interface, was added to OS/2 with Release 1.1. PM supports multiple processes running within a single session. Because all PM programs run in a single session, the user has access to and can view all of the executing processes at one time. The user can direct his attention to a single process without switching sessions. The Presentation Manager replaces the command line with an easy-to-use point and click graphical interface. Rather than mastering complex commands, the user merely selects an object or list entry, and PM translates the selection into the necessary commands to perform the task.

The Presentation Manager controls the keyboard, mouse and screen within its session to ensure effective sharing of these resources among its executing processes. PM, in turn, allows indirect access to these resources in a controlled and orderly manner to prevent any program from interfering with or damaging the operation of another program running within the session.

The Presentation Manager session is the default session within OS/2; that is, the PM session is automatically made the foreground session when OS/2 is started. But, as noted earlier, the PM session is not the only one that may be running within an OS/2 system. OS/2 allows multiple full screen sessions to run simultaneously with the Presentation Manager session, and when the user leaves the PM session and switches to one of the full screen sessions, all the standard session restrictions still apply (see Figure 1-4).

PM thus acts as a mini operating system, providing selected services to those applications running within its session. Applications still must rely on the OS/2 Kernel to supply system-wide resources. To support this requirement, PM applications can call the OS/2 Kernel directly via the Dos service calls. But, because PM controls the processes executing within its session, applications that wish to run in the PM session must be designed and written to special PM standards.

A Word About Presentation Manager Windows

The Presentation Manager uses the concept of windows to effectively share the keyboard, mouse and screen. The window is the basis for allocating user input and

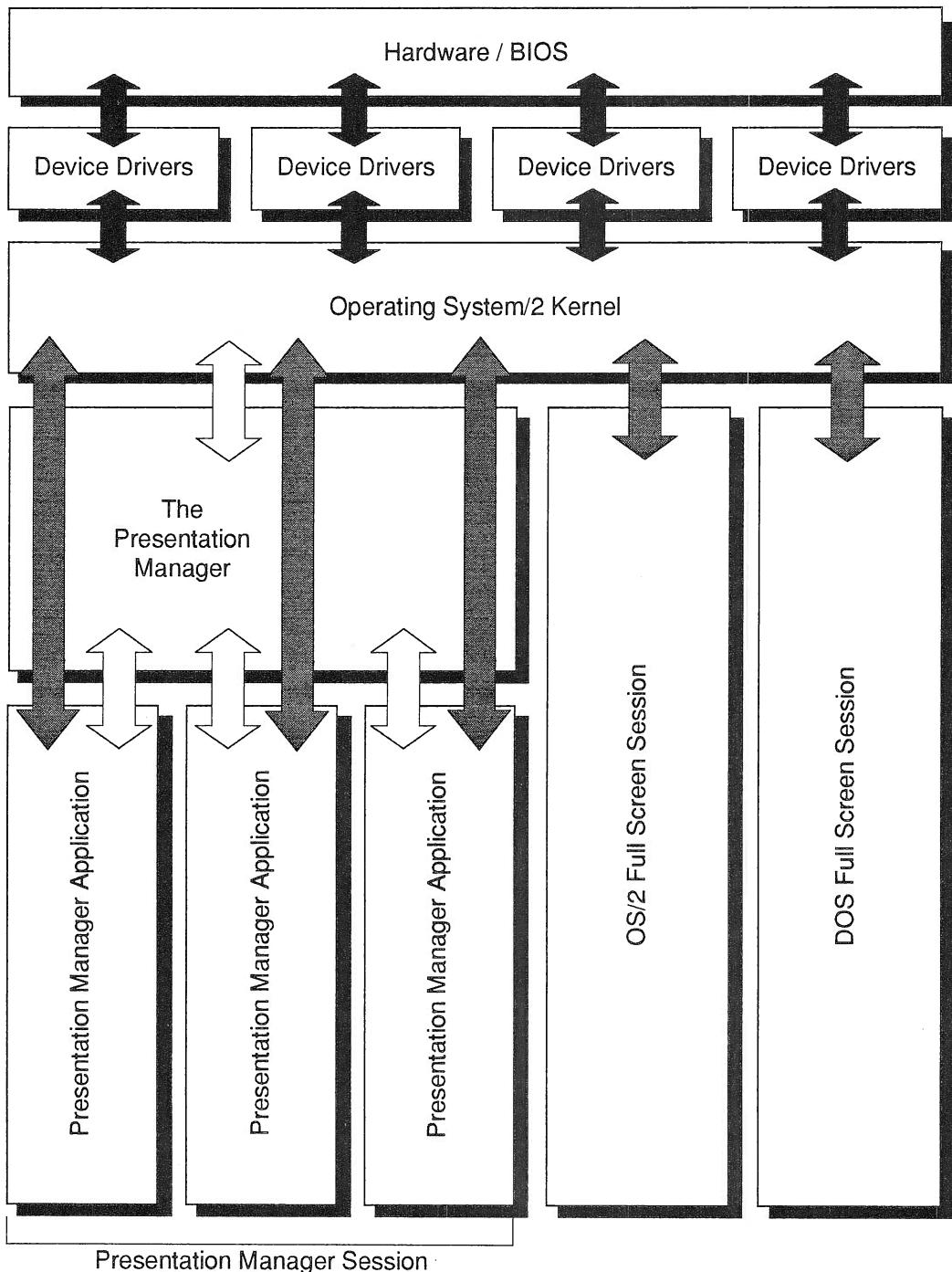


Figure 1-4 The OS/2 - Presentation Manager environment

controlling display output. A user may only communicate with a program through one of its windows. A window is a rectangular space bounded by a border, called a frame, with no particular style or appearance. A window's appearance and contents are entirely under the control of the owning process. A window may contain other windows within its frame, such as user control windows, child windows and dialog windows.

A program's window and its position relative to all other windows on the screen determines which program will receive the user's input and what data the user can see. Only the focus window, that window on the top of all other windows on the screen, will receive user input. The focus window is said to have the focus, and at any given point any window may have the focus. The user can control which window has the focus by moving the mouse to the desired window and clicking within that window's frame, or by going directly to the desired process via the Task List. The order in which the windows are stacked on top of each other is called the z-order (see Figure 1-5). By definition, the window at the top of the z-order has the focus. The z-order of windows is not related to how and when windows are created, only to their arrangement along the depth axis (the

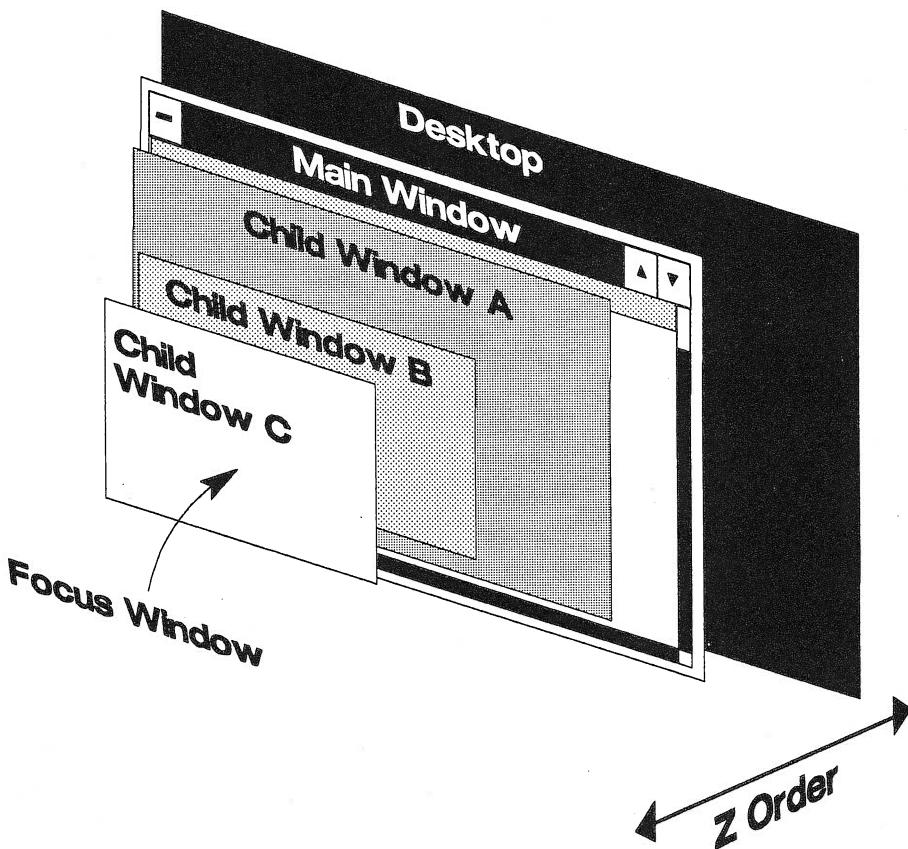


Figure 1-5 The z-order of windows

z-axis) at any point. The z-order of the windows is constantly changing as the user moves from window to window.

A Word About Window Classes

Every PM window must belong to a window class. Multiple windows may share the same class, but every window must belong to a class. A window class associates the code procedure that will process messages for the window (known as the window procedure) and the style (look and feel) of the window under a common name. This is part of the object-oriented nature of the Presentation Manager. For every window there must be an associated set of code that is executed whenever the user interacts with the window.

There are two groups of window classes, public window classes and private window classes.

Public window classes are predefined styles and window procedures that are available to all PM programs. The window procedures to support these predefined classes are contained in PM dynamic link libraries. PM supplies several preregistered public classes referred to as control window classes. Here are the more common PM control window classes. Those marked with an asterisk are supported under Version 2.0 only.

CONTROL WINDOW CLASSES

WC-BUTTON	Button or box windows that may be selected by clicking the pointing device or using the keyboard.
WC-COMBOBOX	An entry field control and a list box control merged into a single control.
WC-CONTAINER*	A control that holds a collections of objects.
WC-ENTRYFIELD	A control that accepts a single line of text.
WC-FRAME	A composite window class that may contain other window classes as children.
WC-LISTBOX	Presents a list of items within a scrollable window from which the user may select one or more of the items.
WC-MENU	Presents a list of items within a horizontal window from which the user may select one item.
WC-MLE.	A window that accepts and displays multiple lines of text.
WC-NOTEBOOK*	A control that provides single access to multiple groups of controls.
WC-SCROLLBAR	A window that allows the user to request scroll movement of the contents of a related window.
WC-SLIDER*	A control that allows the display and user control of a value through the use of a slider arm.

WC-SPINBUTTON	A single line entry field with associated up and down arrows that allow the user to alter the contents of the entry field.
WC-STATIC	Display items that do not respond to the pointing device or keyboard interaction.
WC-TITLEBAR	A window displaying a title and allowing movement of the owner window.
WC-VALUESET*	A control that allows the user to make a single choice from a collection of bit maps, icons, colors, text or numbers.

Private window classes are established by a process for its own use and may not be shared with other processes. A private window class must be established by the process prior to the creation of the first window to use the class. For private window classes, the process must supply both the code procedure and the style information. Window classes are established by passing a class name, class style flags and the window procedure address to PM as part of a function referred to as registering the class. Then, when individual windows are created, the class name is passed to PM, allowing the correct connection to the window procedure and class style to be made.

Multiple windows with the same style and function can be supported by a common window procedure. It is not uncommon to find multiple windows with the same style and function associated with a single window class. Each window class, but not each window, is required to have unique window procedure to support its operation.

Establishing A Window's Look And Feel

Every window has certain behavioral characteristics that are collectively defined as a window style. The window style determines the behavior of the window and how it will interact with other windows on the display screen. If, for example, you want the user to be able to see a window, you would select a window style of WS-VISIBLE. There are eleven window style flags that control the appearance and behavior of an individual window. When a window is created, the binary sum of one or more of these flags is passed to PM to set the style of the window.

WINDOW STYLE FLAGS

WS-CLIPCHILDREN	Prevents a window from painting over any of its children.
WS-CLIPSIBLINGS	Prevents a window from painting over any of its siblings.
WS-DISABLED	A window cannot receive user input.
WS-GROUP*	This control window is part of a group.
WS-MAXIMIZED	The window is created maximized.

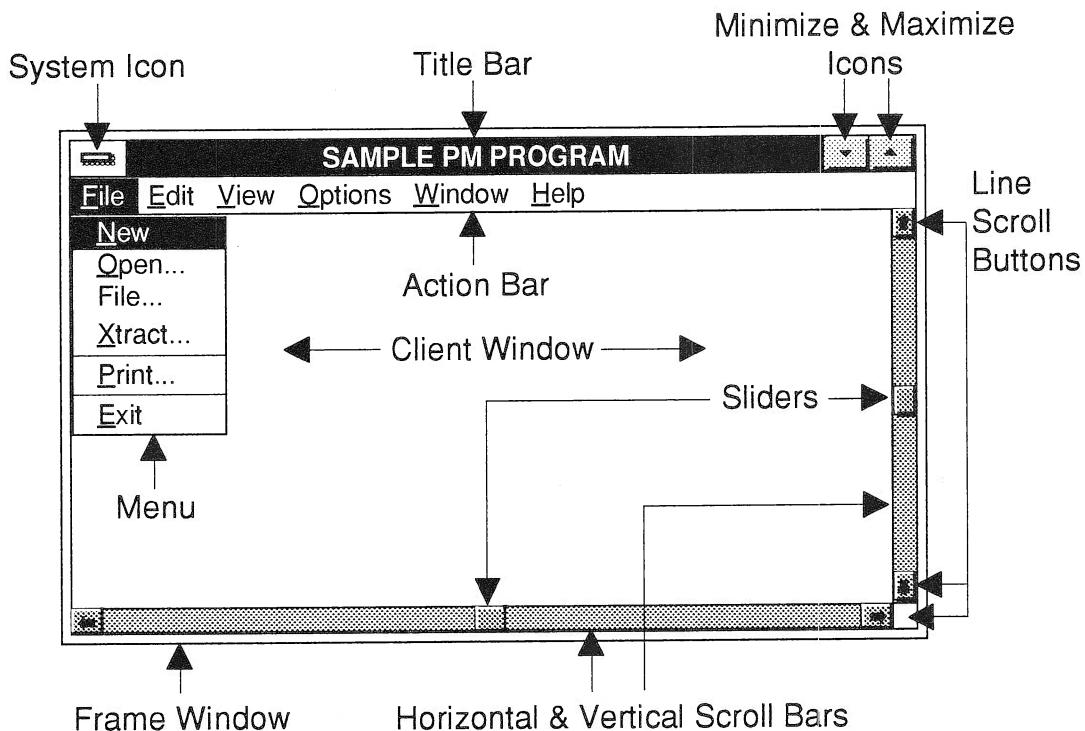


Figure 1-6 The individual parts of a window

WS-MINIMIZED

The window is created minimized.

WS-PARENTCLIP

Clips a window to the boundaries of its parent.

WS-SAVEBITS

Saves the underlying screen image when this window is made visible.

WS-SYNCPAINT

The window is synchronously repainted.

WS-TABSTOP*

When the Tab key is pressed, the cursor will stop at this control window.

WS-VISIBLE

The window is created visible.

*Applies to dialog boxes only.

Just as there are individual window styles, there are styles that apply to entire classes of windows, called a class style. Class styles function in much the same way as window styles, but they define the behavior of all the windows belonging to the class. For example, if you wanted each window in a class redrawn when it is altered in size, you would specify a class style of CS-SIZEREDRAW. There are ten different window class styles flags; one or more of these styles must be assigned to a class when the is registered.

CLASS STYLE FLAGS

CS-CLIPCHILDREN	Prevents a window from painting over any of its children.
CS-CLIPSIBLINGS	Prevents a window from painting over any of its siblings.
CS-FRAME	The window has the properties of a frame window.
CS-HITTEST	When the mouse moves over the window a hit-test message is sent to the window before the normal mouse movement messages to see if the window procedure wants to receive mouse movement messages.
CS-MOVENOTIFY	A move message is sent to the child windows when the parent window is moved.
CS-PARENTCLIP	Clips a window to the boundaries of its parent.
CS-PUBLIC	Registers this class as a public class available to any process.
CS-SAVEBITS	Saves the underlying screen image when this class is made visible.
CS-SIZEREDRAW	The window is redrawn whenever it is repainted.
CS-SYNCPAINT	The window is synchronously repainted.

What appears to be a single complex window on the display screen is usually not a single window but a combination of many windows, called control windows, each with its own specialized functions. A window like the one in Figure 1-6 is composed of many specialized controls. Which control windows are included within a frame window and how the frame window will react to a user request for moving and sizing is determined by the frame create flags. As with other flags, the binary sum of one or more frame create flags are passed to PM when the window is created.

FRAME CREATE FLAGS

FCF-ACCELTABLE	Include a keyboard accelerator table with this window.
FCF-BORDER	Place a border around the window.
FCF-DLG BORDER*	Place a dialog box border around the window.
FCF-HORTSCROLL	Include a horizontal scroll bar in the window.
FCF-ICON	Use a window icon when the window is minimized.
FCF-MAXBUTTON	Include a maximize button in the window.
FCF-MENU	Include an action bar in this window.
FCF-MINBUTTON	Include a minimize button in the window.
FCF-NODYTEALIGN	Do not ensure that the window is always on an 8 pel boundary.
FCF-	The window is not moved when its owner window is moved.
NOMOVEWITHOWNER	Allow PM to determine the size and position of the window when it is first made visible.
FCF-SHELLPOSITION	

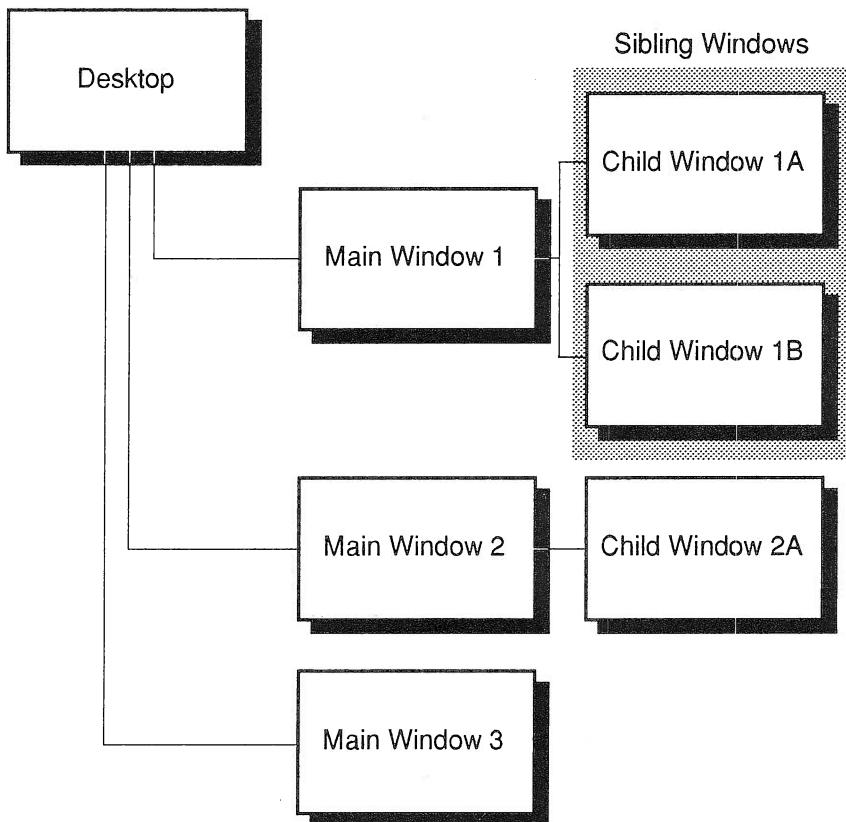


Figure 1-7 The hierarchical arrangement of windows (program view)

FCF-SIZEBORDER

Make the window sizable.

FCF-SYSMENU

Include the system menu in the window.

FCF-SYSMODAL

The frame window is system modal.

FCF-TASKLIST

Insert the name of this program into the Tasklist menu when this program is first started.

FCF-TITLEBAR

Include a title bar in the window.

FCF-VERTSCROLL

Include a vertical scroll bar in the window.

* Applies to dialog boxes only.

The Window Hierarchy

The hierarchical relationship of one window to another is very much like the relationship of individuals within a family. There is a single window called the desktop at the top of the hierarchy. The desktop is owned by PM. Every window created has a parent - child

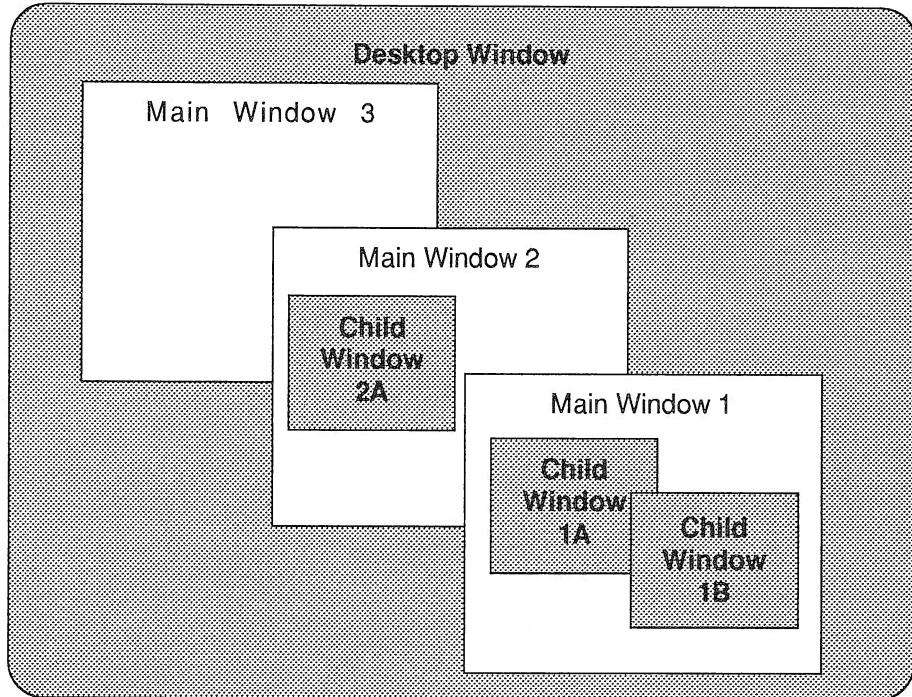


Figure 1-8 The hierarchical arrangement of windows (user view)

relationship with the desktop and the other windows created by a process. This relationship is important because a parent window's boundaries determines the movement and sizing limits of all of its descendants. The first window created by a program is called the Main window. All windows created from the Main window are children of the Main window, and all windows created out of these children are children of the Main window's children. This hierarchy can continue indefinitely as windows are created. When more than one window is created with a common parent, these windows are called sibling windows. When a window is created with another window's child as its parent, that child window becomes the parent of the newly created window. Thus, a window may at the same time be a parent window, a child window and a sibling window. All this may seem unnecessary, but it is the relationship between windows that determines the boundaries for movement and sizing of windows. As an example of how important this hierarchy is, if five levels of windows are created with the Main window as their common ancestor, none of those windows may be moved or sized beyond the boundaries of the Main window as they are all children of the Main window. See Figures 1-7 and 1-8 for additional views on window hierarchies.

Window Ownership

Just as every window has a parent, every window is owned by a window procedure. I say owned by, but what I really mean is that the window procedure supporting the owner window has special authority over the created, or client, window, receiving notification of significant events affecting the window and exercising control over the window. The term owner window comes from the fact that during window creation, the handle of a window, representing the window procedure that is to exercise control over the newly created window must be passed to PM. This handle allows PM to connect the correct window procedure to the newly created window.

The concept of an owner is especially important when developing dialogs that use control windows to act as clients for the owner window. These windows are often the focal point of user input and they must be carefully controlled by the owning window procedure to keep the user within the desired bounds of input. Control over these windows is given to the owning window procedure by authorizing it to send command messages to the dialog or control window.

Using this messaging scheme, the owning window procedure can engage in a conversation with the client window receiving notification of significant events, taking action based upon the notifications, controlling the type and sizes of data accepted by the client window, and sending and extracting data from the control window. Without this ability to react to user input through the system of messages, the complexity of building client windows would be greatly enlarged.

The Message-Based PM System

Since the Presentation Manager controls all input and output within its session, it is important to understand how commands and data flow within the Presentation Manager session.

All information flows between the Presentation Manager and applications in the form of messages. All messages within the PM session -- and there are hundreds of different message types -- have the same simple format, the handle of the window that is the target of the message, the message ID, two message parameters, a message time stamp and the cursor location at the time the message was generated. Here is the COBOL definition of the Presentation Manager message structure. This structure must appear in every COBOL PM program that you write.

```

01  QMSG .
  05  QMSG-HWND          pic 9(9) comp-5.
  05  QMSG-MSGID         pic 9(4) comp-5.
  05  QMSG-PARAM1        pic 9(9) comp-5.
  05  QMSG-PARAM2        pic 9(9) comp-5.
  05  QMSG-TIME          pic 9(9) comp-5.
  05  QMSG-POINT .
    10  QMSG-X            pic 9(9) comp-5.
    10  QMSG-Y            pic 9(9) comp-5.

```

In addition to defining the message structure for use with messages received from external sources, you must define the message structure within the program's Linkage Section for use by each window procedure. Messages may only be passed to a window procedure via the Linkage Section. Here is the Linkage Section definition for the PM message structure. This structure must appear in every COBOL PM program that you write.

```

01 hwnd                      pic 9(9) comp-5.
01 Msg                       pic 9(4) comp-5.
01 MsgParm1                  pic 9(9) comp-5.
01 Redefines MsgParm1 .
  05 MsgParm1w1              pic 9(4) comp-5.
  05 MsgParm1w2              pic 9(4) comp-5.
01 MsgParm2                  pic 9(9) comp-5.
01 Redefines MsgParm2 .
  05 Msgparm2w1              pic 9(4) comp-5.
  05 MsgParm2w2              pic 9(4) comp-5.
01 MsgTime                   pic 9(9) comp-5.
01 MsgXPointer               pic 9(9) comp-5.
01 MsgYPointer               pic 9(9) comp-5.

```

All of the message fields with the exception of the message ID field are defined as 4-byte long integers. The message ID field is defined as a 2-byte short integer. We will be examining each of these fields in detail later, but here is a very basic description of each of the message fields.

QMSG-HWND

A handle is a shorthand identification for the window procedure that is to receive this message.

QMSG-MSGID

The number that identifies this message.

QMSG-PARAM1 & QMSG-PARAM2

Message parameters 1 and 2 contain all of the data being passed within the message. The usage and structure of the data within these parameters varies with each message. It is important to check the data structure of each parameter when working with messages. To correctly extract the data from a message, you must know the data structure of each parameter.

QMSG-TIME

This field contains the time that the message was generated.

QMSG-POINT

The field contains the mouse location in X,Y coordinates at the time the message was generated.

The Flow Of Information Within The Presentation Manager

Messages within the Presentation Manager session are generated in one of three ways.

- 1) The user initiates an action with the mouse or keyboard.
- 2) A PM call made by a procedure indirectly generates additional messages.
- 3) A procedure sends or posts a message to a procedure in another thread or to another procedure in its own thread.

User interaction with the Presentation Manager is by far the most common source of PM messages. Figure 1-9 shows the flow of these messages through the PM system. User interaction causes one or more messages to be built by PM and placed into the System Message Queue. As messages are placed into this queue, they are removed from the bottom by the System Router and after the keyboard scancodes are translated into the correct ASCII characters using the current codepage, the message is passed to the correct application. Before placing the message into the Application's Message Queue, the message is tested for accelerator key translation or mouse hit testing, both functions unique to each application, and translated if required. The message is then placed into the Application Message Queue. As messages are placed into this queue, they are removed from the bottom by the application's WinGetMsg call, which waits for the arrival of a message if the queue is empty when the call is issued. Based upon the message handle, the message is then routed to the correct procedure within the application using the WinDispatchMsg call. Finally, the message is processed by the procedure or passed to the PM default procedure for default processing, if this message is of no interest to the procedure.

Applications generate a significant number of messages by taking action on behalf of a window, such as creating, destroying, resizing, moving, repainting, or drawing into a

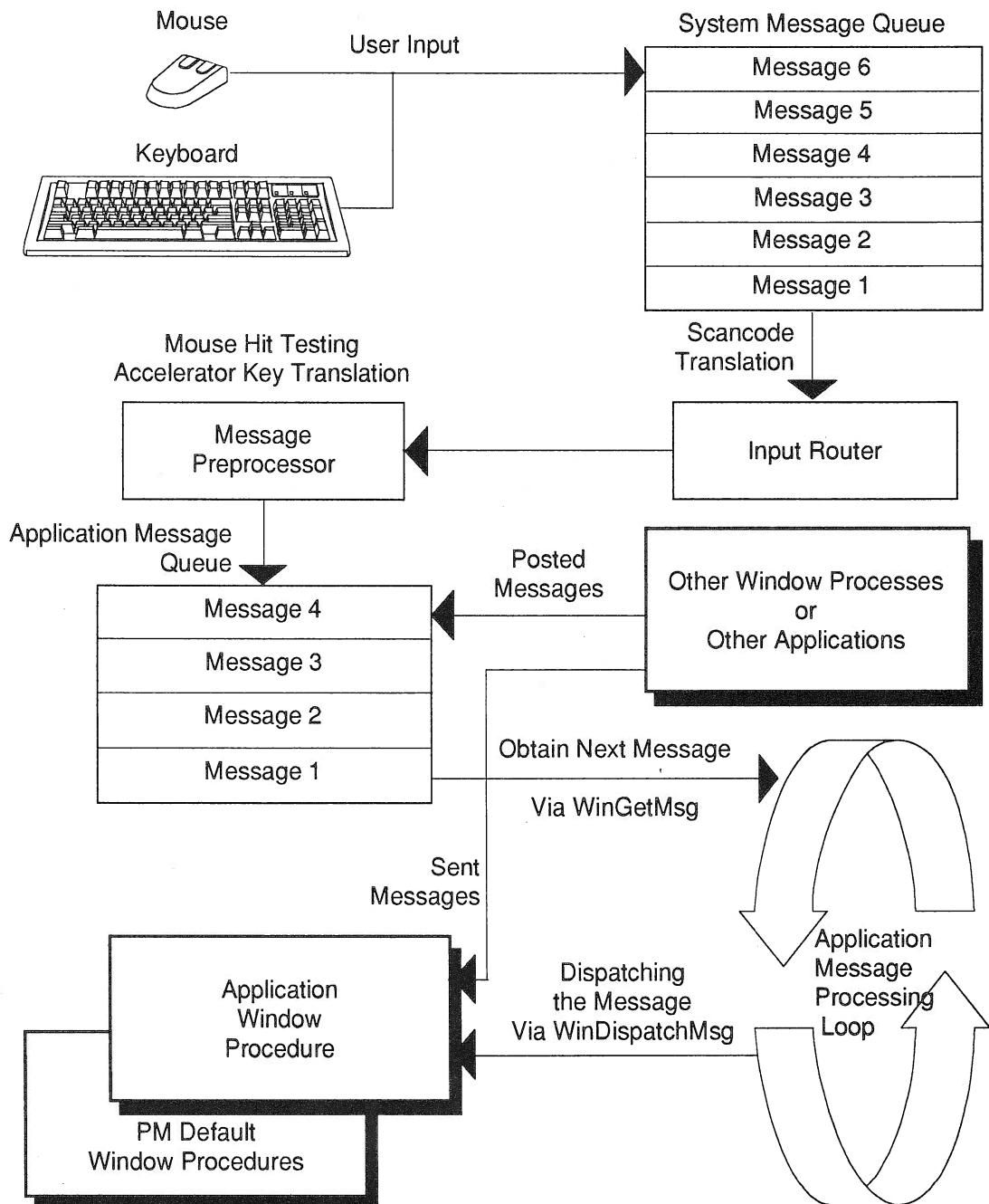


Figure 1-9 The Presentation Manager message processing loop

window. Actions such as these generate additional system messages that flow through the system in an identical manner to user-generated messages.

Messages are also generated when a procedure explicitly sends or posts a message directly to a procedure in another thread or another procedure within its own thread. We will take a closer look at the difference between sending and posting messages in a later chapter, but for now remember that sent messages go directly to the window procedure and are not placed into the Application's Message Queue. The procedure that sends the message is suspended until control is returned by the target procedure. Posted messages are placed into the target Application's Message Queue, where they are processed like any other arriving message. When posting messages, the originating procedure is not suspended and continues to execute while the posted message is processed. If synchronization is required between procedures when using posted messages, an external device like a semaphore must be used.

As detailed in Figure 1-9, message processing looks rather straightforward. Messages are generated, flow to the correct procedure and are processed. But this simplistic diagram hides a real coding challenge caused by recursive messages. Recursive messages occur when a procedure takes an action that generates one or more additional messages that are dispatched back to the same procedure while the original message is still being processed. The possibility that more than one message will be active within a single procedure at one time requires that all procedures be reentrant, with specific attention paid to how and where data is saved. The use of flags and common save areas can be extremely dangerous when used in a recursive environment. There are special procedures supported by both the COBOL compiler and PM itself to assist in managing values within a recursive environment, as we shall see in later chapters.

OS/2 Version 2.0

As this book goes to press, IBM Is preparing for the general availability of a major upgrade of OS/2, Version 2.0. OS/2 Version 2.0 offers a stunning array of new and revised features and functions that should make this version of OS/2 the operating system of choice for almost every personal computer user. Every aspect of OS/2 has seen some amount of change, from its new memory management techniques to new DOS and Windows emulation to a completely new graphical user interface.

But with every major operating system upgrade there are the inevitable concerns about the compatibility of existing programs, the impact on applications under development, the viability of compilers and toolkits and the effect on long range-directions based upon the existing operating system. For OS/2 developers programming in COBOL these concerns include the viability of the COBOL Presentation Manager interface. Since COBOL is not IBM's preferred OS/2 language, there is always the possibility that a new

version of the operating system will discontinue or alter the PM programming interface required by COBOL programs. These are valid concerns; IBM's own COBOL/2 bindings were never enhanced after their introduction with OS/2 Version 1.2 and are no longer a viable PM option. But, based upon initial Version 2.0 Beta code there appears to be only good news for COBOL programmers and developers in OS/2 Version 2.0.

In this section I will try to address the impact that OS/2 Version 2.0 will have upon COBOL Presentation Manager programming and detail what is required to convert your OS/2 Version 1.3 development environment to Version 2.0. To help you understand what changes you will need to make in the way you program the Presentation Manager, this section includes information about the major changes in Version 2.0 that will affect how Presentation Manager programs must be written. This discussion doesn't cover everything that is new in Version 2.0, only those changes that directly affect PM programming. For a more complete, in-depth look at how OS/2 Version 2.0 operates, I recommend the *Application Design Guide* and the three *Programming Guides* that IBM is offering as part of the OS/2 Version 2.0 Developer's Toolkit.

The Flat Memory Model

A significant change with Version 2.0 is the way that OS/2 manages its memory. OS/2 Version 1.3 manages memory using a segmented structure based upon the memory management support of the Intel 80286 processor. This structure, called the segmented memory model, uses a 16-bit address offset that limits individual instructions to references of 64KB or less. As a result, the segmented memory model requires any program code and data sections larger than 64KB to be divided into blocks, or segments, of not more than 64KB bytes. While the COBOL compiler creates these segments and generates the instructions to manage the segment and offset address registers, it is the program's responsibility to correctly manage dynamically allocated segments and to ensure that individual data variables are contained within single segments.

OS/2 Version 2.0 implements a flat memory model structure that uses the 80386 processor's 32-bit address offset to access a single, continuously unbroken address space of 4GB. Since the Intel 80386 processor still requires the use of segment registers, the flat memory model utilizes single code and data segment registers to mark the beginning of the address space. However, Version 2.0 uses 32-bit instruction addresses to allow individual instructions to reference up to 4GB. With a continuous address space of this size there is no longer a requirement to break a program into artificial segments. The elimination of program segments has real benefits for the user and programmer.

- Elimination of the need to reload the segment registers whenever a task switch occurs, or a program moves from one segment to another, significantly reduces a program's execution time. The greater the number of segments in a program,

the larger the potential reduction in execution time. In terms of time, reloading the segment register is a very expensive process.

- The flat memory model simplifies memory management for the application. There is no longer a requirement to fit data into 64KB segments or ensure that the compiler generates and includes the necessary segment crossing code within the program.
- Programmers no longer need to deal with addresses composed of segments and offsets. All flat memory model addresses are composed of a single 32-bit value.
- The flat memory model is supported using the compiler's small memory model. Programmers no longer need to deal with the complexities imposed by the large or huge memory models.

The Process Address Space

Using the flat memory model, OS/2 Version 2.0 allocates a system address space of 4GB and a separate process address space of up to 4GB for each process. While each process can theoretically access up to the 4GB limit, for compatibility with existing 16-bit OS/2 segmented programs, each process is limited to using the bottom 512MB of the address space, referred to as the compatibility region. The address space above 512MB, referred to as the system region, is currently reserved for use by 32-bit system code. Future releases of OS/2 will open this space to use by 32-bit protect mode programs.

Within the 512MB compatibility region, the top 64KB of space is reserved for shared memory objects. Shared memory objects contain dynamic link library routines that may be used by all processes so the address space that hold this shared code is set aside in each process address space. This leaves a maximum of 446MB of memory available for each process's code. But a process's program code (referred to as private code) is only guaranteed the bottom 64KB of the address space. The program must compete with the shared memory objects for the remainder of the address space. As shared memory objects increase, their address space expands downward and as the program code increases, its address space expands upward. These two spaces cannot overlap and if they meet, an out-of-memory condition occurs. See Figure 1-10 for the layout of a process address space.

Memory Management

Version 1.3 allows programs to allocate additional memory, but only in variable-sized segments up to a limit of 64KB. If a program requires more than 64KB of additional memory, it must allocate more than one segment. But, allocated segments carry two significant restrictions because compiler-generated segment crossing code is not available for dynamically allocated memory. For allocated segments, no individual variable or array can exceed the size of the allocated segment (64KB), and no variable or array can span a segment. While variable-sized segments are easy for programmers to create, they are a nightmare for OS/2 to manage.

Version 2.0 standardizes the allocation of memory on a fixed-size memory page, while removing the 64KB limit on allocated memory. Version 2.0 allows a program to allocate additional shared or private memory from 1 byte to the current limit of the process address space (446MB minus the addresses already allocated). But unlike OS/2 Version 1.3 that allocates variable-sized segments, Version 2.0 allocates its memory, called memory objects, as a series of fixed-sized 4KB pages allocated on 4KB boundaries. Any

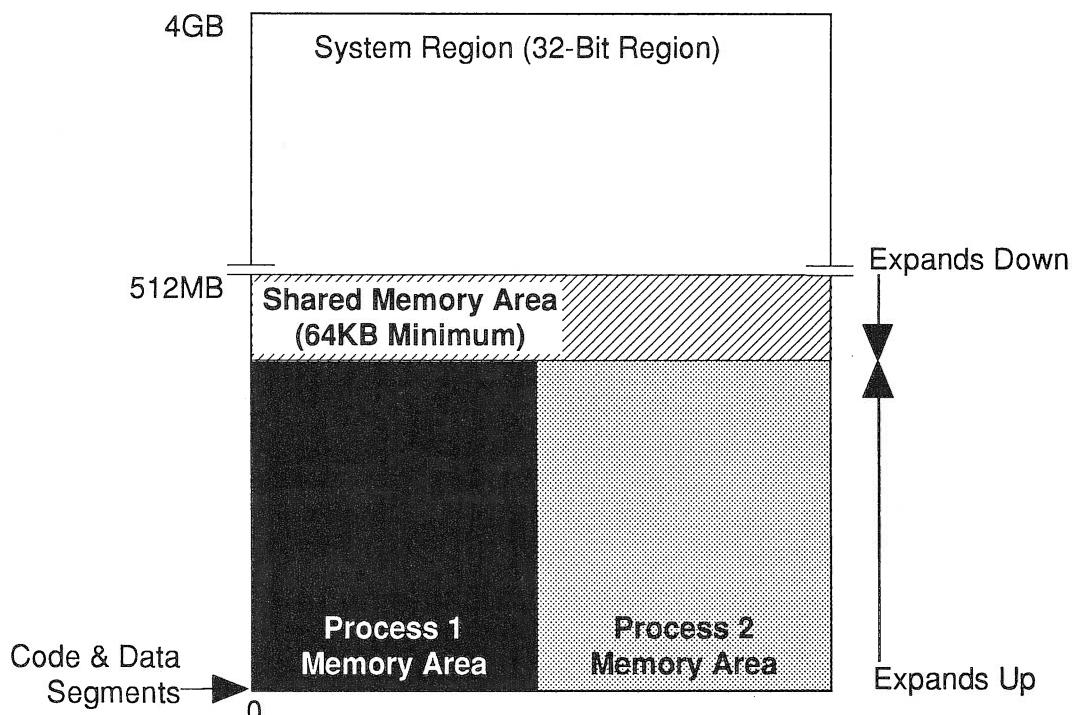


Figure 1-10 OS/2 Version 2.0's process address space

allocation of memory less than a multiple of 4KB will result in the addresses from the top of the allocated memory to the next 4KB boundary remaining unused. For example, if the program allocates 5KB of memory, OS/2 will utilize two pages of memory to hold the allocation resulting in 3KB of memory being wasted. The size of a memory object is set when it is allocated and may not be changed once allocated.

The use of fixed-sized pages can result in a significant loss of memory if the program is not careful to fill each allocated page. Version 2.0 offers a solution that allows a program to allocate a full page or multiple pages of memory, then suballocate the memory into multiple memory objects as needed. To accomplish this, a program initially allocates all the memory it will require as a multiple of 4KB pages. Allocation of memory has no effect on real memory as allocation only reserves the requested addresses in the process address space. Then as memory is required by the program, the amount needed is committed by the program, which causes OS/2 to remove the memory from the free memory pool and add it to the process's real memory. Memory no longer required by the program must be decommitted so that the associated real memory can be returned to the free memory pool. As with memory allocation, memory is committed in 4KB pages and always on a 4KB boundary. While passing variables that span memory objects can cause problems and should be avoided, Version 2.0's ability to allocate extremely large memory objects should allow programmers to avoid this problem.

Page Management

In a similar fashion to the current 1.3 version of OS/2, Version 2.0 treats the addresses generated when a program is compiled as pointers to tables that, in turn, point to the actual memory where the code or data currently resides. By relating program addresses to tables that remain in memory, OS/2 can manage the real memory, moving parts of programs around in memory and even swapping unused parts of a program out to disk, then updating the table pointers to reflect the current location of the page of code or data.

For OS/2 Version 1.3, the unit of memory management and protection is the variable-sized segment. For OS/2 Version 2.0, the unit of memory management and protection is the fixed-sized, 4KB page. Version 2.0 divides a program's 32-bit offset address into pointers, to a page directory entry in the Page Directory Table, to a page table entry in a Page Table and to the offset address within the memory page that holds the required code or data. See Figure 1-11 for a description of how the 32-bit offset address is used to find the required code or data.

The Page Directory contains an entry for each allocated Page Table for the currently active process. When a task switch occurs, OS/2 loads the Page Directory with Page Table information for the process that is about to run. Bits 22 through 31 of the generated address are used as a pointer to the correct Page Table entry in the Page Directory. This

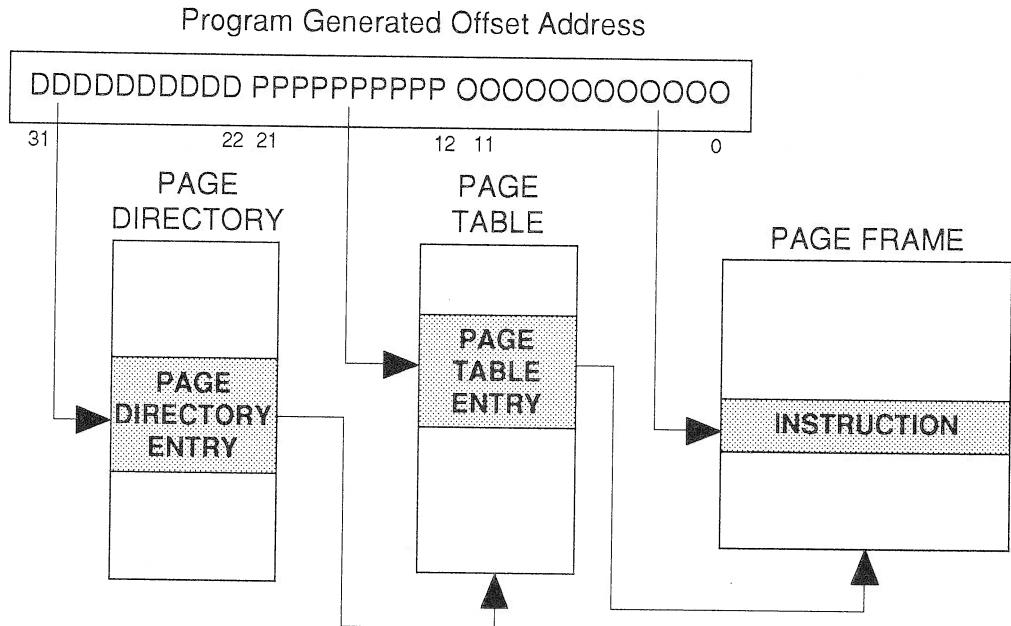


Figure 1-11 OS/2 Version 2.0's address resolution

entry points to one of the Page Tables allocated to this process and bits 12 through 21 of the generated address points to the Page Table Entry within the Page Table. The Page Table Entry points to the memory page that holds the instruction or data, and bits 0 through 11 of the generated address points to the location within the memory page where the instruction or data resides. While this process seems very complex, the 80386 processor performs much of this process using a hardware feature called the Translation Lookaside Buffer. The Translation Lookaside Buffer holds the page addresses of the 32 most recently used pages allowing the TLB to resolve references to these pages without software intervention. See Figure 1-11 for a description of how memory addresses are resolved in Version 2.0.

The Page Directory and each Page Table are held in individual pages. As a result, the Page Directory contains pointers to a maximum of 1024 Page Tables. Each Page Table contains pointers to a maximum of 1024 page frames. Thus, 1024 Page Tables x 1024 page entries in each table x 4KB for each page yields the process address space limit of 4GB.

The Guard Page

Under OS/2 Version 1.3, a program is required to know when an allocated segment is consumed and ensure that it does not address beyond the upper limit of that segment. If the program tries to address beyond the boundary of the allocated segment, a Trap x'00D' is generated and the program terminated.

OS/2 Version 2.0 implements a mechanism called a guard page that removes the requirement that programs remember when its allocated memory is consumed. The guard page is the uppermost page of committed memory for a private memory object and the lowest page of committed memory for shared memory object. When the program accesses memory within the guard page, OS/2 generates a guard page exception that can be trapped by the process and used to trigger the commitment of additional memory.

The Memory Management Process

When a program attempts to access an instruction or data that is not currently in memory, a page fault exception is generated and OS/2's memory management routine is entered to find available memory and load the missing code or data. OS/2 first scans the page frame array to find a page of memory that is currently free. It then loads the required code or data from a library or the Swapper dataset into the allocated page of memory, inserts the page frame address into the page table, sets the present bit on and restarts the program at the instruction that causes the page fault.

OS/2 16-Bit Memory Compatibility

To support existing 16-bit segmented memory processes, Version 2.0 simulates the segmented memory model within the 16-bit program's process address space using a feature called LDT tiling. Through LDT tiling, Version 2.0 maintains a table of up to 8192 descriptors that map the 64KB segments of the segmented memory model on top of the flat memory model's address space. The Local Descriptor Table simulates the segments used in the 16-bit addressing scheme, allowing 16-bit addresses to be resolved to the proper page of memory. The memory manager places each simulated segment on a 64KB boundary and maps the segment onto 4KB pages up to the 64KB limit of the segment. The minimum amount of real memory required, therefore, for any segment allocation is one page. If the requested segment is less than 64KB, the memory addresses from the top of the highest allocated page to the next 64KB boundary are unused. While

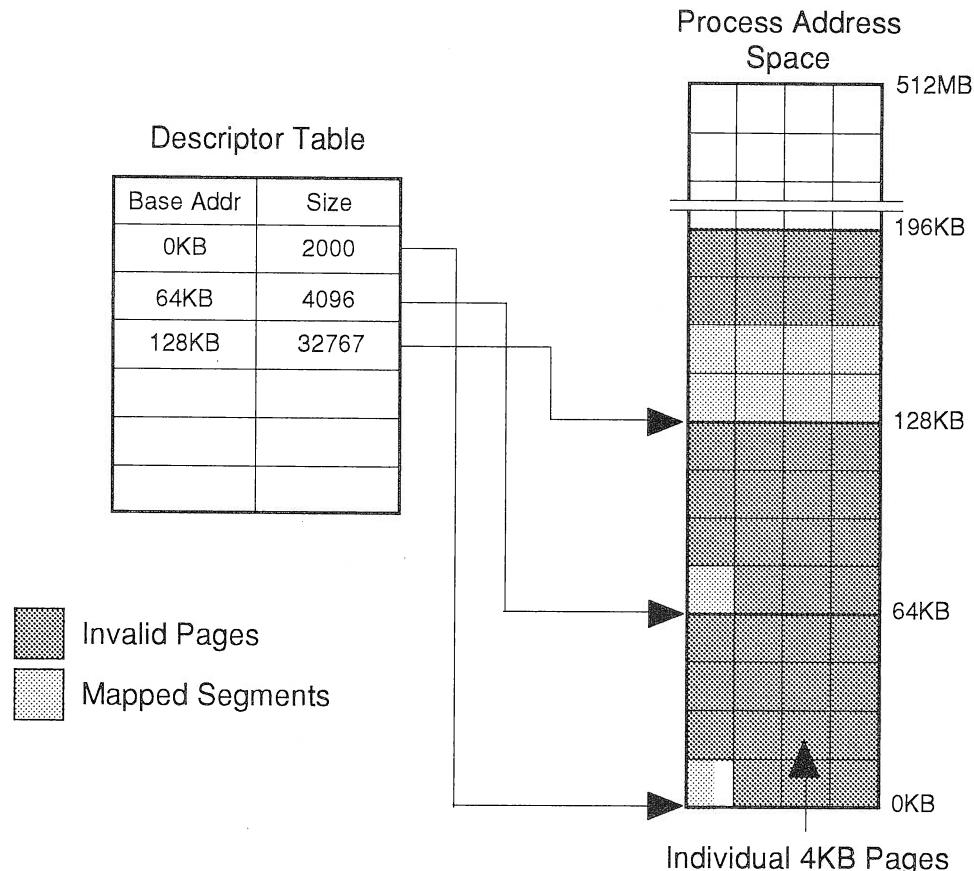


Figure 1-12 LDT Tiling

the actual management of the segments is slightly different in Version 2.0, the differences are hidden from the programs and do not affect program compatibility. See Figure 1-12 for a representation of LDT tiling under Version 2.0.

Thunk Layers

There are basic inconsistencies between the structure of 16-bit programs and the structure of 32-bit programs. These inconsistencies make it impossible for a 16-bit routine to call a 32-bit routine or vice versa without an intermediate code procedure to adjust the inconsistencies. The intermediate code that handles this conversion is called a thunk layer. Version 2.0 contains the 16-bit to 32-bit thunk layers that allow 16-bit programs to call 32-bit OS/2 and PM APIs, and the 32-bit to 16-bit thunk layers to allow

32-bit programs to call 16-bit OS/2 or PM APIs. However, there are no thunk procedures supplied to allow 16-bit programs to access 32-bit DLLs or 32-bit programs to access 16-bit DLLs. The programmer is responsible for placing the correct thunk procedure or procedures into the program making the call or in the program or DLL receiving the call.

A thunk procedure must perform the following functions to allow 16-bit programs or routines to call 32-bit DLLs.

- Convert all addresses passed by the 16-bit calling routine from the 16-bit segment and offset (16:16) to the 32-bit offset (0:32) format used by the 32-bit DLL.
- Convert short integers passed by the 16-bit calling program into long integers used by the 32-bit DLL.
- Realign any structures passed from the WORD alignment used by the 16-bit calling program to DWORD alignment used by the 32-bit DLL.
- Convert the stack from the WORD alignment of the 16-bit calling routine to the DWORD alignment of the 32-bit DLL.
- Convert the far call segment and offset format of the return address pushed on the stack by the 16-bit calling program into the near call offset only format used by the 32-bit DLL.

When a 32-bit DLL returns to the 16-bit calling routine or when a 32-bit routine calls a 16-bit DLL, the thunk procedure must perform the reverse of these same functions. For example, the DWORD aligned stack must be adjusted to a WORD aligned stack. In addition, when going from 32-bit to 16-bit, the thunk procedure must ensure that no single variable or array spans the 64KB segment boundary limit. Variables or arrays that do must be moved to ensure that they are contained entirely within a single segment.

The Workplace Shell

OS/2 Version 2.0 replaces the current PM Shell graphical user interface with a new interface called the Workplace Shell. Based upon IBM's 1991 System Application Architecture Common User Access Workplace Environment, the Workplace Shell transforms the user interface from its current list, menu and command-based format to an object-oriented environment. The Workplace Shell is designed to better shield the user from the complexities of OS/2 and simplify the tasks that the user must perform to a greater extent than the current PM Shell. The Workplace Shell should make OS/2 easier to use, especially by the less experienced user.

The Workplace Shell allows the user to accomplish their tasks through the direct manipulation of screen objects (icons) that represent the item upon which the task is to be performed. For example, to print a report, the Workplace Shell allows the user to select the icon representing the document and drag it to the icon representing the printer, to start editing a document by dragging the document's icon over the icon representing the editor and releasing it, or erase an object by dragging the object's icon to the shredder. All of this is in contrast to the OS/2 File Manager's print and copy commands that are currently required to perform the same functions.

The object is the key to using the Workplace Shell. Objects can be programs, documents, folders, files, physical devices or items defined by the user. Objects are manipulated by clicking on them or dragging and dropping them on other objects. Clicking on a program object, for example, executes that program, clicking on a folder opens that folder displaying the objects inside and clicking on a document executes the program associated with the document and specifies the document as input.

The Workplace Shell introduces four new window controls plus file and font dialogs that make complex dialogs easier for the user than those in Version 1.3. Here are the new Workplace Shell controls. See Figures 1-13 through 1-17 for examples of the new controls and dialogs.

CONTAINER	A control window that logically groups objects (icons)
NOTEBOOK	A representation of a multiple page notebook designed to step users through complex dialogs. Each page of the notebook is a dialog.
SLIDER	A sliding bar that allows the setting of a value by sliding the bar rather than entering a numeric value.
VALUE SET	A group of mutually exclusive choices similar to radio buttons but represented by graphical icons.
FILE DIALOG	A prepackaged set of dialogs that allow a user to display, select and save files using a standard interface.
FONT DIALOG	A prepackaged set of dialogs that allow a user to display, select and implement different fonts.

Compatibility With Version 1.3 Programs

The number one concern of programmers looking at OS/2 Version 2.0 must be its ability to run programs developed under prior versions of OS/2. Not only COBOL programs developed in-house but the thousands of spreadsheets, word processor and graphics programs being used by current OS/2 users. No matter how good OS/2 Version 2.0 is, if it can't run existing OS/2 and DOS programs, then in most cases it can't be justified.

To fully understand program compatibility, you must understand the four types of programs, called programming models, that can be run under Version 2.0. COBOL program compatibility depends upon the programming mode being used.

Pure 16-bit programs. These are programs written as 16-bit programs, compiled using a 16-bit compiler and linked into a 16-bit executable using 16-bit libraries. These programs issue only 16-bit OS/2 and PM API calls and call only 16-bit DLLs. Included within this group are all reentrant COBOL PM programs developed under existing versions of OS/2. Pure 16-bit reentrant COBOL programs are 100 percent compatible with Version 2.0 and will execute under Version 2.0 without modifications. Samples of these types of COBOL programs including the sample program have been run without modifications, recompiling or relinking using OS/2 Version 2.0 Beta code. Please note however that PM programs written in non-reentrant COBOL using an interface layer such as the IBM COBOL/2 Bindings have not been tested and may not run under Version 2.0.

Mixed 16-bit programs. These are programs written as 16-bit programs but compiled using a 32-bit compiler and linked into a 32-bit executable using 16-bit libraries. These programs issue both 16-bit and 32-bit APIs. Compatibility for mixed 16-bit reentrant COBOL programs depends upon the external routines that are called. Mixed 16-bit reentrant COBOL programs that call OS/2 and PM APIs or 32-bit DLLs are compatible and will run under Version 2.0. Mixed 16-bit reentrant COBOL programs that call 16-bit DLLs will not run without a user-supplied thunk procedure.

Mixed 32-bit programs. These are programs written as 32-bit programs, compiled using a 32-bit compiler and linked into a 32-bit executable using 32-bit libraries. These programs issue both 16-bit and 32-bit API calls. Compatibility for mixed 32-bit reentrant COBOL programs depends upon the external routines that are called. Mixed 32-bit reentrant COBOL programs that call OS/2 and PM APIs or 32-bit DLLs are compatible and will run under Version 2.0. Mixed 32-bit reentrant COBOL programs that call 16-bit DLLs will not run without a user-supplied thunk procedure.

Pure 32-bit programs. These are programs written as 32-bit programs, compiled using a 32-bit compiler and linked into a 32-bit executable using 32-bit libraries. These programs issue only 32-bit API calls. Compatibility for pure 32-bit reentrant COBOL programs depends upon the external routines that are called. Pure 32-bit reentrant COBOL programs that call OS/2 or PM APIs or 32-bit DLLs will run under Version 2.0. Pure 32-bit reentrant COBOL programs that call 16-bit DLLs will not run without a user-supplied thunk procedure.

Based upon these four programming models, here is the level of COBOL program compatibility that you can expect for each programming model.

- Reentrant COBOL PM programs developed under OS/2 Version 1.2 or Version 1.3 are 100 percent compatible with Version 2.0 and will run without modification.
- Mixed 16-bit or 32-bit reentrant COBOL PM programs will run only under the tightest restrictions. Unless the compiler's toolkit supplies the necessary thunk procedure or calling code that allows 16-bit programs to call 32-bit programs and vice-versa, mixed model COBOL programs should be avoided.
- Pure 32-bit COBOL PM programs are 100 percent compatible with Version 2.0, assuming that your COBOL compiler produces 32-bit reentrant code and includes a full 32-bit library.

Running Current Compilers Under Version 2.0

As with other existing OS/2 programs, current COBOL compilers should run under Version 2.0 without modification. Existing development and debugging tools should also run, but only for use with 16-bit programs. Version 2.5.39 of the Micro Focus COBOL/2 compiler, used to develop the sample code in this book, has been run under ship level OS/2 Version 2.0 code, without errors.

Migrating Existing Programs to Version 2.0

Before discussing migration of your existing PM programs, a word of caution. Version 2.0 runs only on computers using the Intel 80386 or 80486 processor. The 32-bit programs built for OS/2 Version 2.0 will not run on computers using the Intel 80286 processor. If your company has a significant number of 80286 machines still in production, you may wish to continue developing 16-bit applications until the majority of 80286 processor-based machines are removed from service.

Clearly, the ability to move from your current COBOL PM development environment to the 32-bit development environment rests upon the availability of a COBOL compiler that produced reentrant 32-bit code, the compiler's associated 32-bit libraries and a toolkit that supports 32-bit programs. For programmers and developers with access to these tools, it will be business as usual for Version 2.0. Most PM APIs are supported in both 16-bit and 32-bit modes, and the development of new 32-bit programs should follow the same process currently used. Allowing for some changes in the Kernel's memory management APIs and new exception handling APIs, all of the PM structures presented in this book are valid for OS/2 Version 2.0.

The majority of the problems that will experience converting existing COBOL PM programs to Version 2.0 will occur with the Kernel calls, those that start with Dos. The Kernel calls account for the majority of the changed APIs, as you would expect. With most of the Version 2.0 changes involving Kernel functions, this is where you will see most of the required modifications. Conversely, if your programs use few Kernel calls, you should see relatively few required changes.

The lack of a thunk procedure and the inability to write thunk procedures in COBOL mean conversions to 32-bit code should be limited to the pure 32-bit programming model to guarantee success. To ensure the pure 32-bit programming model, converted 16-bit programs may not call OS/2 or PM APIs that are not supported in 32-bit mode or call DLLs or subroutines written in 16-bit mode, compiled using a 16-bit compiler or linked using 16-bit libraries. The single exception is calling 16-bit PM or OS/2 Kernel APIs. Calling any of these 16-bit APIs will, by definition, make the program a mixed 32-bit model, but because OS/2 and PM have the necessary thunk layers built in, these programs should run. A word of warning, however: Anything included in OS/2 for compatibility, such as 16-bit APIs, is subject to being removed in future releases of Version 2.0.

To determine which PM or Kernel APIs need to be replaced when converting to pure 32-bit code, you should compare the APIs used in each 16-bit program against the list of PM and Kernel calls in Appendix E. Depending upon the type of call you are comparing, the following applies.

For PM calls, check each call against the PM calls listed in the Appendix. If the 16-bit PM call appears in the left-hand column with a 32-bit call opposite it in the right-hand column, as in the following example, the 16-bit call must be converted to the 32-bit call shown.

DosPrintJobGetInfo	SplQueryJob
--------------------	-------------

If no PM call appears in the right-hand column, as shown in the following example, then the 16-bit call must be replaced with a different 32-bit call or calls. This will usually involve changes in the supporting code as well as the individual call.

PrfCreateGroup	
----------------	--

PM calls that appear only in the right-hand column are new 32-bit calls and may be used to replace 16-bit calls that are no longer supported. Be aware however, that substituting new PM calls that are not direct replacements for existing 16-bit PM calls may involve a significant amount of change to the supporting code.

Calls that do not appear in either column are supported in both 16-bit and 32-bit format, and do not need to be replaced.

For OS/2 Kernel APIs, the procedure is slightly different. Appendix E contains all of the OS/2 Kernel APIs so you should always find an entry for an existing 16-bit Kernel API. Here, if an API appears only in the left-hand column, as in the following example, it must be replaced with an OS/2 API that appears only in the right-hand column.

DosFSRamSemClear		
------------------	--	--

An API that appears in the left-hand column with an API opposite it in the right-hand column, as in the following example, must be replaced by the API shown in the right-hand column.

DosCWait		DosWaitChild
----------	--	--------------

Finally, calls that appear only in the right-hand column are new 32-bit APIs that have no 16-bit counterpart. These APIs may be substituted for 16-bit APIs, but the warning issued above still applies. Inserting a new 32-bit API that is not a direct replacement for an existing 16-bit API will involve some amount of redesign and recoding to the API's support routine.

I should also warn you that just because the replacement 32-bit API has the same name as the existing 16-bit API, it may not be coded the same. Always check the structure of any new APIs added to your program to be sure the structure is correct.

Here is the routine to follow when making the changes required to convert a 16-bit reentrant COBOL program to a pure 32-bit reentrant COBOL program.

- Replace any 16-bit OS/2 or PM APIs not supported under Version 2.0 with their new 32-bit versions. This may involve changes to the code that supports the 16-bit API. Existing OS/2 and PM APIs supported under Version 2.0 do not need to be replaced.
- For any 16-bit API that does not have a corresponding 32-bit function and thus no 32-bit replacement API, recode the section of the program containing the 16-bit API adopting a new 32-bit function. For example, there is no segment memory allocation under Version 2.0 so any program using the DosAllocSeg call must change not only the API but method of memory allocation used.

- Replace all 16-bit DLLs with new 32-bit DLLs. The inability to code thunk procedures within a COBOL program ensures that 32-bit COBOL programs cannot call 16-bit DLLs. At a minimum, converting the DLL will involve recompiling and relinking, but be prepared to change one or more of the APIs within the DLL. Examine the actual call to the DLL as well. Changing the DLL to a pure 32-bit model may require changes to the data passed by the call.
- Recompile and relink the program using a reentrant 32-bit COBOL compiler and 32-bit library. The 32-bit OS/2 and PM library is named OS2386.LIB.

For every change required when converting a 16-bit program, there are other changes that should be made to use the new features of Version 2.0. While these changes are not required for your converted programs to run, they should be considered when upgrading existing programs to make them more compatible with the features and functions of Version 2.0. Here are some areas that you may want to consider changing when converting a 16-bit program to a 32-bit format.

- Make the program responsive to direct manipulation. Support drag and drop functions and data association.
- Add cut and paste support if it is not already included.
- Replace, where appropriate, existing dialogs with the new Workplace Shell dialogs.
- Replace existing interprocess communications with Version 2.0's newer facilities.
- Make changes to memory allocation routines to cut potential wasted memory.
- Enlarge internal arrays and structures beyond 64KB where beneficial.
- Convert structure and array entries to long integers, and ensure structure and array alignment is DWORD.

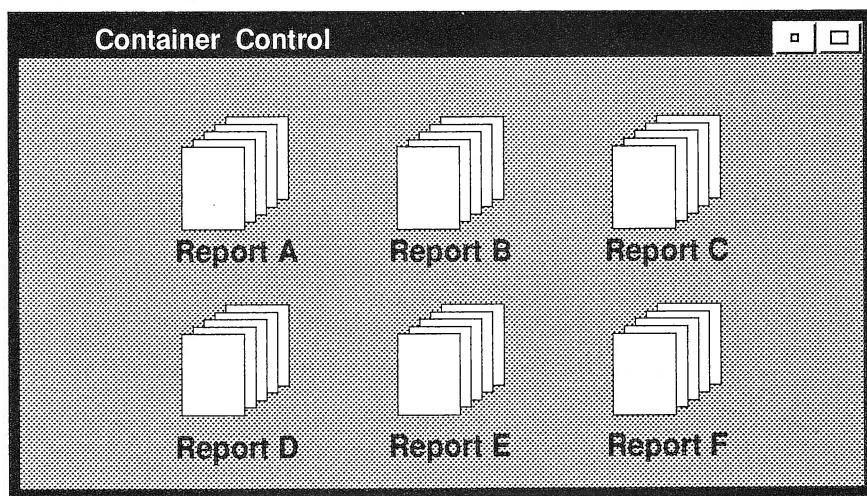


Figure 1-13 The Container window control

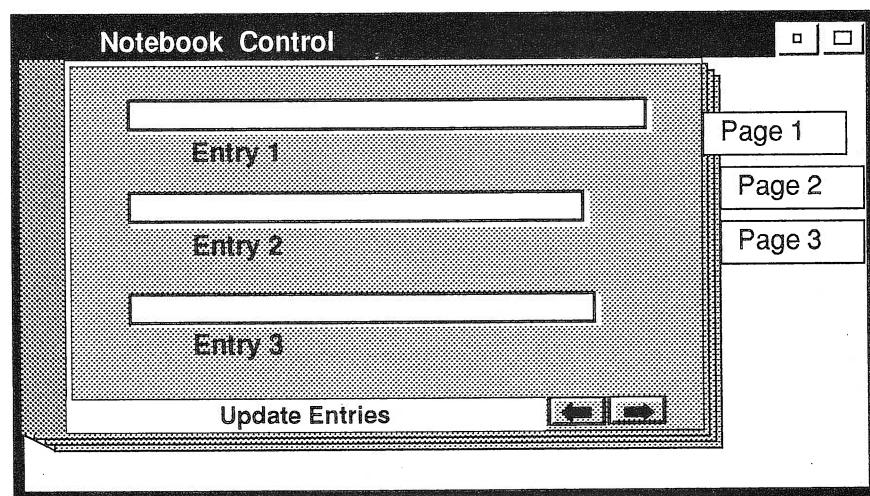


Figure 1-14 The Notebook window control

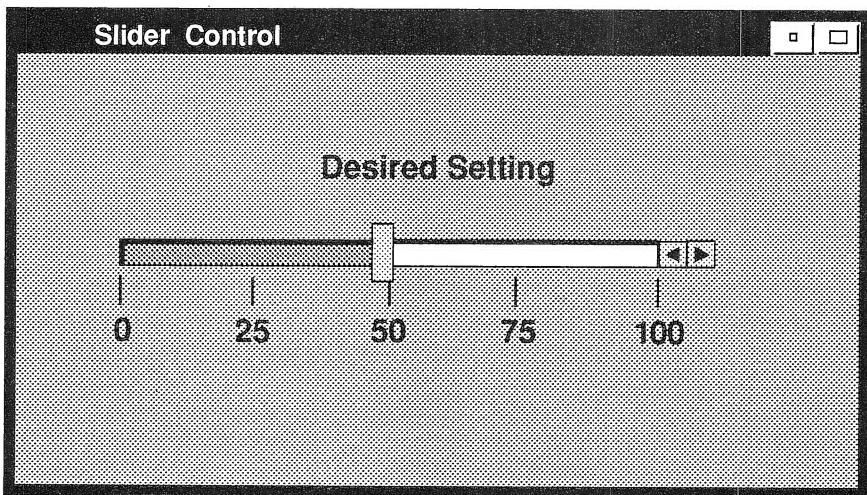


Figure 1-15 The Slider window control

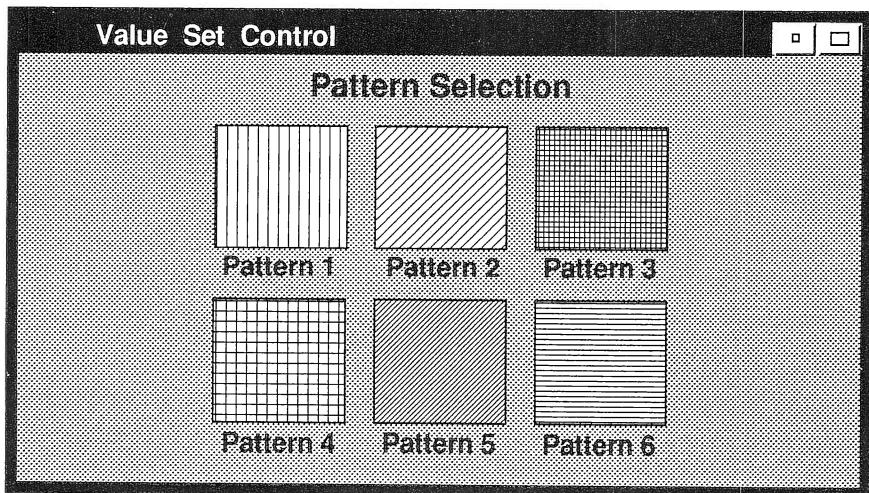


Figure 1-16 The Value Set window control

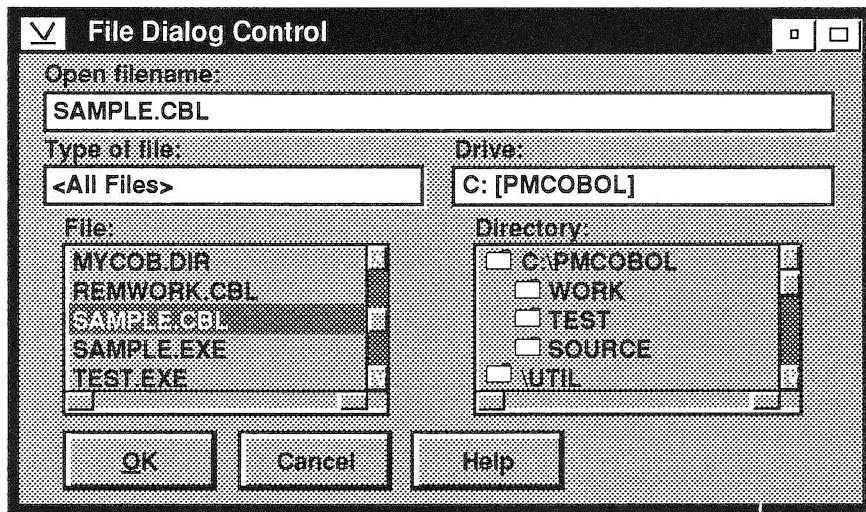


Figure 1-17 The File System dialog control

Chapter 2

Using COBOL With The Presentation Manager

Regardless of the hardware platform, most COBOL programs have traditionally followed a consistent, highly structured form with a top to bottom logical flow through the code. Born in the traditional host batch environment, COBOL programs have always executed by following the order of the logic laid down by the program's designers. The program developers determine the order of every event that can occur during the program's execution. Once started, a COBOL program executes automatically, following the rigid paths set down by the programmer. The user has no control over the order of program execution. If the user is to be involved in the execution, it is only at the point where the programmer determines user input is necessary and includes the logic to support such input.

COBOL programmers have spent years learning these structured programming techniques, only to be faced with developing programs for OS/2 and PM that have a very different structure and absolutely no predetermined order of execution. For certain, the hardest adjustment you will need to make during your transition to PM programming is to abandon COBOL's formal top to bottom structured form of programming in favor of a program structure that is totally subservient to its execution environment. This is really a new mind-set. It is not the Presentation Manager calls, the new terminology or the message-based system that will cause you problems, but the changes in program design brought about by this new execution environment that will cause you the most problems.

Of course, having to code to the whims of the user is a humbling and frightening experience, to say the least. Considering the following requirements that PM places on its programs, you can have some real headaches to contend with when designing and coding a PM program.

- PM programs do not control the order of their execution. As a result, programmers can no longer rely on the contents of save areas, buffers, flags and I/O areas

because a programmer can no longer be sure of the order of events that lead to this point in the logic.

- PM programs must be recursive; that is, be able to issue calls that result in messages to itself. This will result in multiple logic threads being active within the same code sections simultaneously.
- PM programs must be fully reentrant to support the PM environment. Programmers must take care to ensure data integrity as there may be multiple occurrences of the same variable active at any time.
- PM programs must remain responsive to the user regardless of where the program is in its logic. Programs may respond to user input in from .1 to .3 second. Where a relatively long time is required for processing, an additional logic thread must be created to allow the user to immediately regain control of the system.
- PM programs have no control over how much, if any, of their output the user can see.

Programming Window Procedures

The solution to these programming problems and the key to constructing PM programs can be found in the concept of Object Oriented Programming. OOP dictates that for every class of window there is a specific code procedure that supports that window. The window and its supporting code procedure are referred to as an object, as neither can function without the other. The window without the code will not function and the code without the window will never execute. The code procedure supplies the logic to support only those functions represented by the window. There is no logic to support interrelationships with other procedures or control the order of window manipulation. Each object executes as if it were in a vacuum, completely independent of any other procedure.

By writing procedures that perform very limited functions with no interdependencies and no external references, a programmer no longer needs to be concerned with the order of execution. When the window is manipulated by the user, the corresponding code procedure is notified, performs the desired functions, then gives up control. Each procedure runs independently, using only the data passed to it in the message, restricting variable data to local storage and returning results via a return code or external data pass area.

These procedures can be simple or complex, depending upon the capabilities offered by the window. A code procedure must contain the code to support at least one function,

but there is no limit to the number of code functions that a procedure can contain. When procedures become complex, the message evaluation loop helps to break a single, large and complex procedure into several tiny procedures, each isolated and independent from the other procedures. Each procedure must contain the following standard message processing logic.

```
entry 'WindowProc' using by value hwnd
      by value Msg
      by value MsgParm1
      by value MsgParm2

Move 0 to MResult
Evaluate Msg

When initialization message
    Perform window initialization functions
    •
    •
    •

When ABC message
    Perform ABC processing
    •
    •
    •

When XYZ message
    Perform XYZ processing
    •
    •
    •

When other message
    Return message to PM for default processing
End evaluate
End Procedure
```

Now, having said no interdependencies and no global save areas, you will probably never see an absolutely pure PM program written in COBOL. Most programs use a combination of restricted global save areas together with local storage work areas. But no matter how you write your code, the key to a good PM program is the amount of procedure isolation. The more isolated the procedures, the less likely you are to have problems during execution. Conversely, the more interdependencies that your procedures have, the more problems you will experience.

An Extended Compiler vs. The COBOL/2 Bindings

There are two different approaches to writing PM programs using the COBOL language. While each approach produces approximately the same look and feel, they utilize different programming philosophies. It is important to understand the differences between these two philosophies.

Extended Compiler: This approach uses a COBOL compiler that has been modified and extended to support C-oriented PM calls, support the Pascal calling convention and produce reentrant programs. By supporting the native PM interface, these compilers produce programs that have no restrictions in their support of the Presentation Manager. Because they use PM's C Language interface, called the C Bindings, programs produced with these extended compilers can perform any function that can be performed by a C Language PM program. On the down side, the coding required for these compilers is somewhat more complex and with a slightly different look than normal COBOL programs. Later in this chapter there is a list of some of the unique coding that must be used if a COBOL program is to run within the PM session.

COBOL/2 Bindings: Rather than modifying the compiler to work with the C Bindings, this approach uses a character-oriented PM interface, called the COBOL/2 Bindings, that accommodates standard COBOL ANSI compilers. The COBOL/2 Bindings, introduced with OS/2 Release 1.2, supports COBOL's character-based data values and normal calling structure. Use of the COBOL/2 Bindings produces a smoother, more standardized code that is very familiar to COBOL programmers.

Most of the changed coding conventions required by the extended compilers are not required when using the COBOL/2 Bindings. However, programs produced using the COBOL/2 Bindings are not reentrant and have major restrictions in their support of the Presentation Manager. Here is a list of the major PM functions that are not available to PM programs written using the COBOL/2 Bindings.

- **Sent Messages:** Non-reentrant programs may not receive synchronously transferred messages (sent messages) from PM or other window procedures. This restriction means that Dynamic Data Exchange, delayed rendering to the Clipboard and the owner draw functions are not available to these programs.
- **Window Subclassing:** Because non-reentrant programs may not receive sent messages, they may not intercept messages sent to other procedures. Intercepting messages destined for other procedures is the basis of window subclassing.
- **The Heap Manager:** This facility for managing memory segments and objects is not available to non-reentrant programs.
- **System Hooks:** Non-reentrant programs may not set system hooks, as this

function requires the ability to receive sent messages.

- Help Manager: Use of the Information Presentation Facility to automatically handle help requests is not available to non-reentrant programs.
- Advanced VIO Calls: Commonly called AVIO, this class of calls supporting PM text windows is not available to non-reentrant programs.

I cannot recommend the use of the COBOL/2 Bindings. The restrictions imposed by the lack of reentrant code more than outweigh the benefits of the character-based interface. In addition, IBM failed to update the COBOL/2 Bindings to include the PM extensions introduced with OS/2 Version 1.3 or Version 2.0, indicating that for all practical purposes the COBOL/2 Bindings has been stabilized at the OS/2 Release 1.2 level. As a result, this book was written for use with an extended COBOL compiler. The sample code used in this book was developed using the Micro Focus COBOL/2 16-bit compiler, and tested using the Micro Focus Xilerator debugging tool.

Using The C Bindings

The use of an extended compiler means that your COBOL programs will be calling the Presentation Manager using the C Bindings, PM's C Language interface. Thus, even though you are programming in COBOL, every call parameter, message, data element and structure passed to PM must conform to the C Bindings. This requires translation from the published C Language notation into the correct COBOL equivalent, ensuring that the exact number of bytes are generated and that the data is stored in the correct notation within each field. To help you make this conversion, Figure 2-1 shows the COBOL equivalents for the most common C Language specifications.

There are thousands of predefined values, everything from message identifiers to return codes, included as part of the C header files and you will need to use these definitions within your COBOL program. Converting these values can be time-consuming so to help I have included most of these values in Appendix C and D. These values are translations of the C Language header files supplied as part of the OS/2 Version 2.0 Developer's Toolkit. As a further aid, many COBOL compilers include conversion programs that translate the C header files to the correct COBOL notation. The Micro Focus COBOL/2 compiler includes the utility H2CPY.EXE that converts the C header file #define statements into COBOL level 78 definitions and the deftype statements into equivalent COBOL data definitions.

If you manually convert the predefined values from the C header files, remember to convert any underscore characters contained in the labels to dashes. COBOL does not allow the use of the underscore characters so every label with an underscore must be

Description	Size	C Definition	COBOL Definition
A single character	1 byte	char	pic 99 comp-5
A single byte treated as 8 bits	1 byte	Bool	pic 99 comp-5
An integer	2 bytes	int	pic s9(4) comp-5
A signed short integer	2 bytes	short	pic s9(4) comp-5
An unsigned short integer	2 bytes	unsigned short	pic 9(4) comp-5
A word treated as 16 bits	2 bytes	Bool	pic 9(4) comp-5
A signed long integer	4 bytes	long	pic s9(9) comp-5
An unsigned long integer	4 bytes	unsigned long	pic 9(9) comp-5
A double word treated as 32 bits	4 bytes	Bool	pic 9(9) comp-5
A far pointer	4 bytes	far	pointer
A near pointer	2 bytes	near	no equivalent
A single precision floating point number	4 bytes	float	no equivalent
A double precision floating point number	8 bytes	double	no equivalent

Figure 2-1 C Language format conversion chart

converted. The Micro Focus H2CPY utility makes this change for you during the conversion.

Implementing the Pascal Calling Convention

Every COBOL compiler that supports non-standard calling conventions should have a statement to specify the Pascal calling convention. The Micro Focus COBOL/2 compiler uses the Calling-Convention statement within the Special-Names paragraph to specify exactly how data should be passed between mixed-language programs. The calling-convention value is a 16-bit number from 0 to 7. The required calling convention for calls to the Presentation Manager is call convention 3, which defines the following data-passing parameters to the compiler.

- Bit 0 = 1: Push values onto the stack using the Pascal convention.
- Bit 1 = 1: Called program will remove parameters from the stack.
- Bit 2 = 0: Return codes will be placed in the RETURN-CODE register.

The Special-Names paragraph is coded as follows; the literal name, OS2API, shown in this example may be any user-defined name.

```
SPECIAL-NAMES.
call-convention 3 is OS2API.
```

Call Parameter Passing

OS/2 uses the hardware stack for parameter passing, so calls to OS/2 and PM can pass only selected data types. For Version 1.3, or any program compiled using a 16-bit COBOL compiler, the following data types may be passed.

A word	-	2 byte value pushed directly onto the stack.
A double word	-	a 4-byte value passed as two, 2-byte words, the low word followed by the high word.
A pointer	-	two, 2-byte words, the offset followed by the selector.

This limits 16-bit program call parameters to short integers, long integers and pointers. This limitation makes the conversion of the many C data types easier. No matter what data type the OS/2 or PM call specifies, you can be assured that it must be converted into one of these three data types for passing with the call. For example, many PM calls specify a data type of Boolean. A Boolean data type is really an unsigned short integer with the contents treated as 16 logical bits rather than as a single value. So, the Boolean value 0110 can be passed as an unsigned short integer with a value of 6.

COBOL programs compiled using a 32-bit compiler and running under Version 2.0 utilize the 80386 registers to pass double words. As a result, the following two data types are the only types utilized by 32-bit programs.

a double word	-	passed as a single 4-byte value.
a pointer	-	passed as a single 4-byte value.

The conversion of C data types for 32-bit programs is even easier than for 16-bit programs. Every passed value is either a double word (long integer) or a pointer.

The Micro Focus COBOL/2 compiler uses two phrases that allow programmers to specify how a call's parameters are to be passed. If a value is to be passed by being pushed onto the stack, the *by value* phrase must be specified. If a pointer to a value or string is to be passed, then the *by reference* phrase must be specified. The question you will face in translating the PM calls defined in C to COBOL is which phrase to use. The selection is easy; here is how to make the choice.

Using the appropriate *Presentation Manager Programming Reference*, determine the parameter's use. It will be defined as:

INPUT:

The parameter's value is being passed from your program to PM.

OUTPUT:

The parameter's value will be returned by PM to your program at the call's conclusion.

INPUT/OUTPUT:

The parameter's value is being passed from your program to PM, but will be updated by PM at the call's conclusion.

RETURN:

This value will be returned by PM as the return code at the call's conclusion.

For numeric values labeled as INPUT, the value must be passed by pushing it onto the stack, so the *by value* phrase must be specified.

For numeric values labeled as OUTPUT or any parameter labeled as INPUT/OUTPUT, a pointer to the data item must be pushed onto the stack, so the *by reference* phrase must be specified.

For alphabetic or alphanumeric strings, regardless of the label, a pointer to the data item must be pushed onto the stack, so the *by reference* phrase must always be specified. Characters can never be placed directly on the stack.

For RETURN, no definition is needed. The returned value will automatically be placed in the Return Code Special Register based upon bit 3 of the calling convention being 0. If the *Returning* phrase is coded as the call's last parameter, the returned value will be copied from the register to the specified variable.

The Use Of Null-Terminated Strings

A major problem for COBOL programs that call the Presentation Manager is the passing of characters. OS/2 and PM, like all C Language programs, do not understand the concept of alphabetic variables. They cannot process alphabetic characters as an entity, with a length and format. C Language programs treat alphabetic variables as a string of ASCII characters, starting at the referenced address and continuing until the first hexadecimal 0, the end-of-string flag, is encountered. There is no length, no sending or receiving variables, no truncation or padding of data, only a starting point and an ending point. Whatever is in between these two points constitutes the character string.

Since you are using the C interface when calling PM, your programs must use the C null-terminated string notation for every character string that is sent to PM and accept null-terminated strings that are sent from PM. Null-terminated strings require a pointer, pointing at the first character in the string and a null-termination character at the end of the string. Failure to append a null-terminator to a character string will cause PM to ripple through memory until a hexadecimal 0 is found or until a storage protection exception is generated.

When defining character string constants that will be sent to PM, simply add a null character after each definition. There are two ways that the null-terminator character can be appended to a string, by concatenation and by redefinition. Here are examples of both methods.

```
01 MessageOne      pic x(17) value "Message Number 1" & x"00".
01 MessageTwo
    05 MsgTwo       pic x(16) value "Message Number 2".
    05 filler        pic x value x"00".
```

Character strings that are received from PM will have the null-terminator appended. This includes character strings sent directly from PM, character strings taken from resource files or data entered by the user. It is important to remember that to COBOL the null character is a valid part of the string and not a special function flag. This means functions such as string compares will include the null character if it is present. As a result, the following character strings will not yield a equal comparison.

COBOL String	"A string"
ASCII value	065 032 115 116 114 105 110 103
PM String	"A string "
ASCII value	065 032 115 116 114 105 110 103 000

As a guide for handling the null-terminator, the following rules should be observed.

- For character strings that will be returned to PM without being processed by your program, leave the null-terminator as a permanent part of the string.
- For character strings that must be processed by your COBOL program, use COBOL's Inspect statement to replace the null-terminator with a blank.

Here is a sample of the Inspect statement to replace the null character at the end of a string with a space. This function works correctly because, by definition, there can be no x'00' characters within a C Language string.

```
Inspect SampleString replacing x"00" by space.
```

Character strings that originate outside PM and must be passed to PM will be your biggest problem, as they do not have the null character required by PM. If you are

familiar with the data, you may be able to use the Inspect statement to add the null character, but for character strings with embedded spaces or strings with unknown contents the Inspect statement probably won't work. If the character strings are for display and do not require content processing, set up a generalized save area, big enough for the largest character string, fill it with spaces, follow it with a null character and lay the variable string into the save area. PM sees the character string as the valid characters followed by one or more spaces. Here is how to code one of these generalized save areas:

```
05 MessageBoxText .
  10 MBoxText          pic x(80) .
  10 filler            pic x value x"00" .
```

If trailing spaces are not an option, you will need to write a string parsing routine to locate the end of the string and add the null character.

Working With Intel Reverse Order Storage Values

When passing values between COBOL programs the structure of the data within individual fields is automatically handled by COBOL. However, when writing programs that exchange data with non-COBOL programs, you must become concerned with the way data is stored. Due to the nature of the Intel microprocessor, computers based upon this architecture store values larger than one byte in a reverse order, with the low byte followed by the high byte. As an example, the integer that holds the value 492 is stored as x"EC01" not x"01EC" as you would expect. Because OS/2 and PM run on machines using Intel microprocessors, data exchanged with OS/2 and PM must use this reverse form of storage.

Fortunately, you will not need to deal with this requirement for a majority of your PM code as most COBOL compilers can store and retrieve values in the Intel reverse order format. The Micro Focus COBOL/2 compiler uses the Computational-5 (COMP-5) identifier to specify data items that must be saved in the Intel reverse order format. As a result, you will see that almost all of the data variables within a Micro Focus PM program have the COMP-5 format identifier.

There are a few OS/2 calls, such as the DosAllocSeg, that require you to understand and work directly with these reverse fields, but they are the exception, not the rule. While you do not need to understand this reverse storage format to code, you will come face to face with this scheme during program debugging and testing. When using debugging tools such as the Micro Focus Xilerator, you will need to fully understand this reverse

order storage notation and be able to enter data directly into the debugging tool in this reverse order format.

Using The Returning Parameter

The Micro Focus Calling Convention 3 specifies that the return codes will be returned in the Return Code Special Register. You can obtain the value directly from this special register or use the Returning parameter, as I have done in the sample programs. The Returning phrase causes the returned value in the register to be copied to the variable specified in the returning phrase. Of course, if you do not want the returned value, you can leave off the Returning parameter and skip access to the Return Code Special Register.

Initializing Numeric Variables

COBOL automatically clears Working-Storage to spaces (x"20"). However, PM expects numeric variables to be initialized to low values. As a result, you must remember to initialize all numeric variables that will be passed to PM to 0. If you do not, any unused variables passed to PM will appear to contain large numbers, 538,976,288 (x"20202020") for a PIC 9(9) variable and 8,224 (x"2020") for a PIC 9(4) variable.

You can initialize these variables by moving 0's to the variable, adding the phrase "value 0" to the variable's definition or, if you are using the Micro Focus COBOL/2 compiler, you can set a compiler directive as follows.

```
$set defaultbyte "00".
```

Static Linked PM Calls

Intraprogram calls or calls made to subroutines that are contained within the object modules that are input to the Linker are resolved during the Linker run. The Presentation Manager APIs, however, are not part of the Linker's input and are not included in the program's executable. PM APIs are contained in OS/2 dynamic link libraries and the external calls made to them by a program must be resolved when the program is loaded. To ensure that this can happen, the COBOL compiler must build a

list of the external references within the program so the Program Loader can find and resolve them at load time. For the Micro Focus COBOL/2 compiler, this list is generated by invoking the static linking feature of the compiler. Invoking static linking for dynamically linked APIs appears to be a dichotomy. But, specifying static-linking to the compiler causes it to build a list of external references. How these references are resolved is the province of the Linker. For dynamically linked modules, the Linker uses the OS2.LIB import library for Version 1.3 and the OS2286.LIB import library for 16-bit programs under Version 2.0 to partially resolve these external references by building place-holders that the Program Loader will resolve at program load time. Static linking with the Micro Focus COBOL compiler is invoked by using the compiler directive /LITLINK or preceding each call with a double-underscore prefix. Either method will produce the same results.

Care should be used with the /LITLINK directive. If your programs call COBOL subroutines, the use of /LITLINK may cause problems with data passing. If your programs do call COBOL subroutines, you may find using the double underscore for PM or OS/22 calls a better choice than the /LITLINK directive.

The compiler command file used to build the sample programs in this book includes the /LITLINK compiler directive. But, for those who wish to use the double-underscore prefix, here is an example of how to code this option.

```
Call OS2API '__WinInitialize' using
      by value ShortNull
      returning hab
```

Working With Handles

Earlier I mentioned objects in connection with Object Oriented Programming. Objects are composed of a window and the code procedure that supports it, treated as a single entity. Since the PM calls that you will be making act upon these objects, PM requires the identify the target object. PM uses a shorthand notation, called a handle, to identify target objects. Think of a handle as the object's internal name. The handle of a window, handle of a message or handle of a file is nothing more than a way to tell PM which window, message or file you are referencing. Actually, handles have been around for some time. File handles introduced with early releases of the DOS operating system serve the same purpose as PM handles.

Actually, within PM just about everything has a handle, not just the traditional objects. Handles are such a convenient way to identify a target that PM has implemented them for every possible target of a call. As a result, everything created, registered, loaded or

invoked, establishes a handle that is returned to the program. From that point on, the program must use the handle to identify the item when making a call that will act upon the item.

For COBOL programs, all handles are defined as signed, long integer variables (4 bytes) stored using the Intel reverse storage notation. So all handle variables are defined as pic s9(9) comp-5.

Some Coding Conventions You Should Use

This is a good place to point out some coding standards that were established for use with the Presentation Manager and which I have adopted for my code. These standards have been carried over from C Language notation and may look strange to COBOL programmers, but they should be used when coding PM programs regardless of the language employed. These standards are not required and your program will work correctly if you do not use them, but they do make your code more readable and help keep everything uniform.

- All labels defined externally to your program or that are OS/2 or PM default definitions should be completely capitalized. Thus, the window paint message should be coded WM-PAINT.
- Labels defined within your program that are composed of multiple names, such as Main-Window-Class, should be concatenated with the first letter of each name capitalized. Thus, Main-Window-Class should be written as MainWindowClass.
- When a label represents a handle, the label should begin with the four letters *hwnd*, always in lower case with the first letter of the following unique name capitalized. Thus, the handle of the main window is written hwndMainWindow.

The Essential Parts Of A COBOL PM Program

Every Presentation Manager program has four basic parts. They are:

- 1) The Initialization routine
- 2) The Message Processing routine
- 3) One or more window procedures
- 4) The Termination routine

Note that I refer to these routines as Presentation Manager routines. These routines are not the same as the COBOL routines with the same name. The PM initialization routine, for example, should not be confused with the usual COBOL initialization routine. These routines may be intermixed or coded separately, but the order of placement of the PM routines within the program is fixed.

The Initialization Routine

This routine registers your program with PM, allows PM to perform the necessary housekeeping and notifies PM of the address of the code procedures that will be used with the various window classes. This routine usually includes the construction of the program's first, or Main Window. At a minimum, this routine must perform the following PM calls and in the order shown (see Figure 2-2). Note however, that this routine usually includes many more housekeeping functions beyond these required four.

- Initialize the PM interface. The WinInitialize call registers your program with PM and returns the Anchor Block handle *hab* to the program.
- Create The PM Message Queue. As soon as your program is registered with PM, it will start receiving messages. To avoid losing these messages, you must establish your program's Application Message Queue, using the WinCreateMsgQueue call immediately after PM initialization.
- Register the window classes. This call must be issued once for each private window class to be used. Registering a window class assigns the class style and code procedure address used by all windows created with this class. Class registration is not required in this routine, but a class must be registered before it can be assigned to a window during creation. Normal coding convention calls for all class registrations to take place within this routine. To register a window class, the entry point is set and the WinRegisterClass call issued.
- Create the Main Window (optional). Following the registration of the window classes, the Main Window should be created. It is usually done within this routine.

The Message Processing Routine

This routine is entered from the initialization routine. The message processing routine removes messages from the Application Message Queue as they become available, using the WinGetMsg call, and routes them to the correct code procedure, using the WinDispatchMsg call. Between these two calls is a test for the message indicating that the program should end. These two calls and single test run in a continuous perform loop until the program is done.

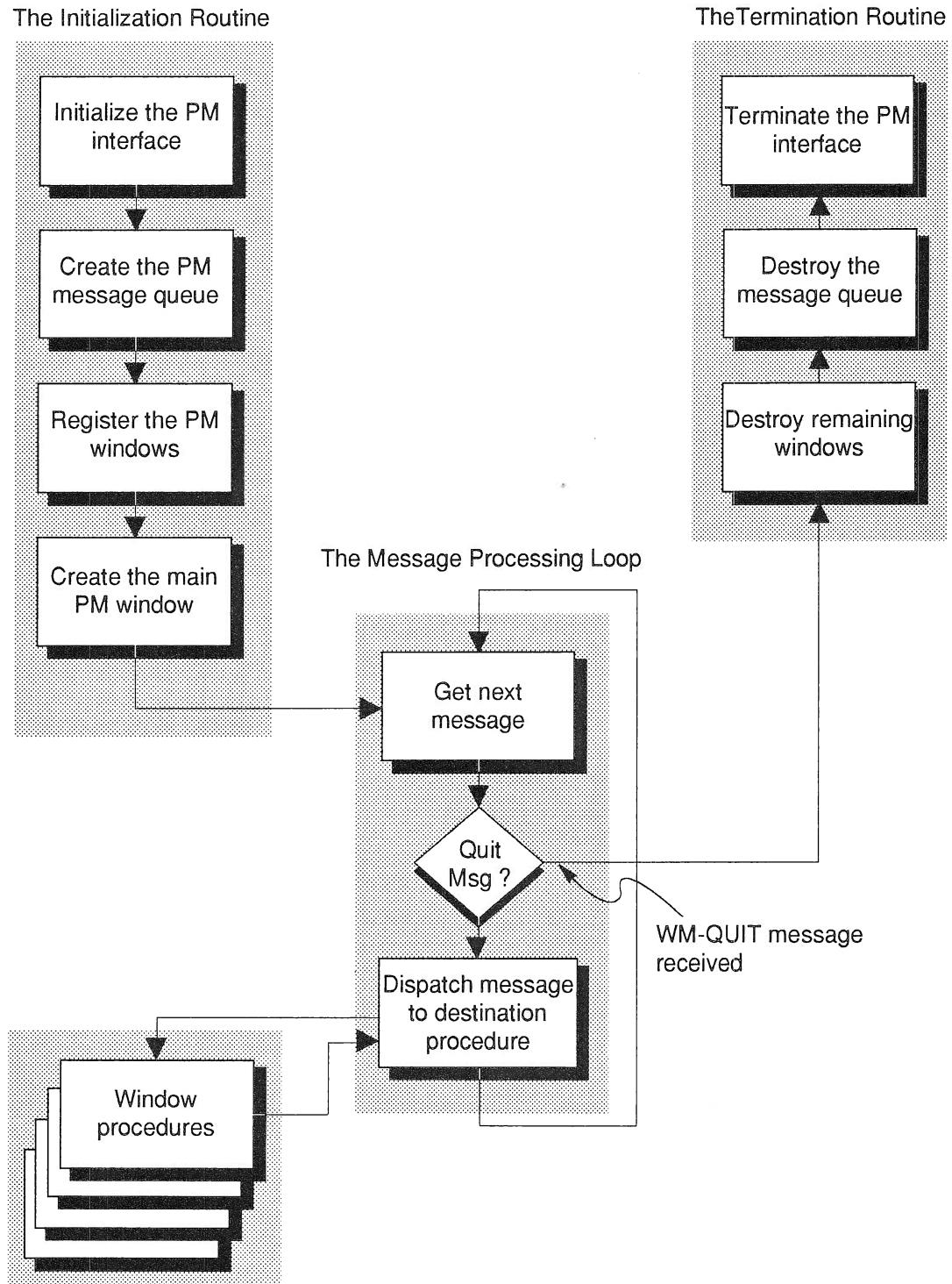


Figure 2-2 The essential parts of a COBOL PM program

Window Procedures

These procedures supply the logic to support each window class. We will examine the building of window procedures in more detail in the following chapters.

The Termination Routine

This routine reverses the functions performed in the Initialization routine, notifying PM to destroy any remaining windows, queues and control blocks and to terminate support for this program. The following three functions form the basis of the termination routine.

- Destroy all remaining windows. Destroying the Main window using the WinDestroyWindow call will automatically destroy all child windows of the Main window.
- Destroy the message queue. Destroying the Application Message Queue using the WinDestroyMsgQueue ends the program's ability to receive messages. This call must be issued immediately prior to the following termination call.
- Terminate the PM interface. The WinTerminate call ends the program's ability to use PM services, but the COBOL program can continue to run after the WinTerminate call.

Chapter 3

Getting Started

Most compilers, workbenches and toolkits have setup procedures that automatically establish and load the subdirectories that they require. The function of leading you through these loading procedures is fairly well automated. Given all of the possible tools that you might be installing, it is impossible for me to guide you through all the procedures. My objective for this chapter is to help you set up the necessary command files to automate the PM program development process and pass along some installation tips I have found helpful.

Setting Up Your Development Environment

Before you begin installing the development products you will be using, make sure you take the following steps.

- Have the installation documentation handy. I know that reading the documentation is tantamount to surrender, but compilers and toolkits have a nasty habit of asking quite detailed questions about your development environment. Not all of these questions are self-evident, and the documentation can steer you around a potential reinstall situation.
- Make a backup copy of your CONFIG.SYS file. While most products you will be installing make backup copies of the CONFIG.SYS file, after installing three or four products you will find a real mishmash of partially altered backed-up versions of the CONFIG.SYS file. Having a copy of the original file is handy.
- Install your compiler before you install the OS/2 programming toolkit. The toolkit will ask for, and try to verify, the library that contains your compiler to set up the sample programs.

- Always allow the product that you are installing to automatically update the CONFIG.SYS file if it attempts to. The OS/2 CONFIG.SYS file is far more complex than the DOS version and the correct entries in the CONFIG.SYS file are absolutely critical. A missed entry or incorrect library definition can result in wasted hours tracking down an obscure compiler, Linker or Toolkit error message.
- In addition to the COBOL compiler and Toolkit directories, you will need a work subdirectory to contain the source code, header files, resource files, bit maps, etc., for the programs you are developing. You may already have a directory you can use. If not, be sure to create one. Mixing your source code with the compiler or Linker can be a real nightmare.

Modifying Your CONFIG.SYS File

As mentioned earlier, you shouldn't need to update your CONFIG.SYS file for most of the products that you install. Any product that requires a manual update to the CONFIG.SYS file should have instructions indicating what updating is required. But, in case you want to review your CONFIG.SYS, here are the required entries for the Micro Focus COBOL/2 compiler or the Micro Focus COBOL/2 Workbench and the IBM Programming Tools And Information Kit (Version 1.3) or the Developer's Toolkit (Version 2.0). I have included only the statements that relate to the COBOL compiler and Toolkit and I have highlighted those parameters that relate to these products. Obviously, your CONFIG.SYS will have additional entries, but you must ensure that these parameters exist as a minimum. If you do not use the Micro Focus products, you will need to substitute the entries required by your COBOL compiler for the COBDIR, COBHNF and LIB entries in the following example. Note, however, that the Linker requires the Toolkit entry in the LIB statement, so if you don't need the LIB statement for your COBOL compiler, be sure that the LIB statement with the Toolkit parameter remains in the CONFIG.SYS file.

Here's a tip. Placing a period followed by a semi-colon at the front of your LIBPATH statement causes OS/2 to search the current directory for the required libraries before searching the subdirectories listed in the LIBPATH statement. Even though you cannot change the LIBPATH during OS/2 execution, placing these characters first in the LIBPATH statement allows you limited manipulation over the library search order by altering the current directory. OS/2 Version 2.0 inserts these two characters into the LIBPATH when it is initially built.

OS/2 Version 1.3

```
LIBPATH=.;C:\TOOLKT13\LIB;C:\COBOL\EXEDLL;C:\OS2\DLL
SET PATH=C:\OS2;C:\OS2\SYSTEM;C:\TOOLKT13\BIN;C:\COBOL\EXEDLL;
SET DPATH=C:\OS2;C:\OS2\SYSTEM;C:\TOOLKT13\IPFC;C:\COBOL;
SET HELP=C:\OS2\HELP;C:\TOOLKT13\BIN;
SET INCLUDE=C:\TOOLKT13\C\INCLUDE;C:\COBOL\INC;
SET COBDIR=C:\COBOL\LBR;C:\COBOL\EXEDLL;
SET LIB=C:\COBOL\LIB;C:\TOOLKT13\LIB;
SET COBHNF=C:\COBOL\ON-LINE
SET IPFC=C:\TOOLKT13\IPFC
```

OS/2 Version 2.0

```
LIBPATH=.;C:\OS2\DLL;C:\COBOL\EXEDLL;C:\TOOLKT20\OS2LIB
SET PATH=C:\OS2;C:\OS2\SYSTEM;C:\COBOL\EXEDLL;C:\TOOLKT20\OS2BIN
SET DPATH=C:\OS2;C:\OS2\SYSTEM;C:\OS2\BITMAP;C:\COBOL;
SET HELP=C:\OS2\HELP;C:\TOOLKT20\OS2HELP
SET INCLUDE=C:\COBOL\INC;C:\TOOLKT20\C\OS2H;
SET COBDIR=C:\COBOL\LBR;C:\COBOL\EXEDLL;
SET LIB=C:\COBOL\LIB;C:\TOOLKT20\OS2LIB
SET COBHNF=C:\COBOL\ON-LINE;
SET BOOKSHELF=C:\OS2\BOOK;C:\TOOLKT20\BOOK
IPFC=C:\TOOLKT20\IPFC
```

The Compile And Link Command File

It always bothered me that C programmers have a magical Make utility that figures out the entire build process and executes it automatically. It bothered me, that is, until I saw the interdependencies that they must deal with just to do a simple compile. COBOL programmers seem to have an easier compile and link process. If you are not managing a significant number of include files, a single command file performs all the steps in the compile, link and bind process. The following example is the compiler command file (MAKE.CMD) used to compile, link and bind the sample COBOL programs. Execute this file by typing MAKE followed by the name of your COBOL source file without the .CBL.

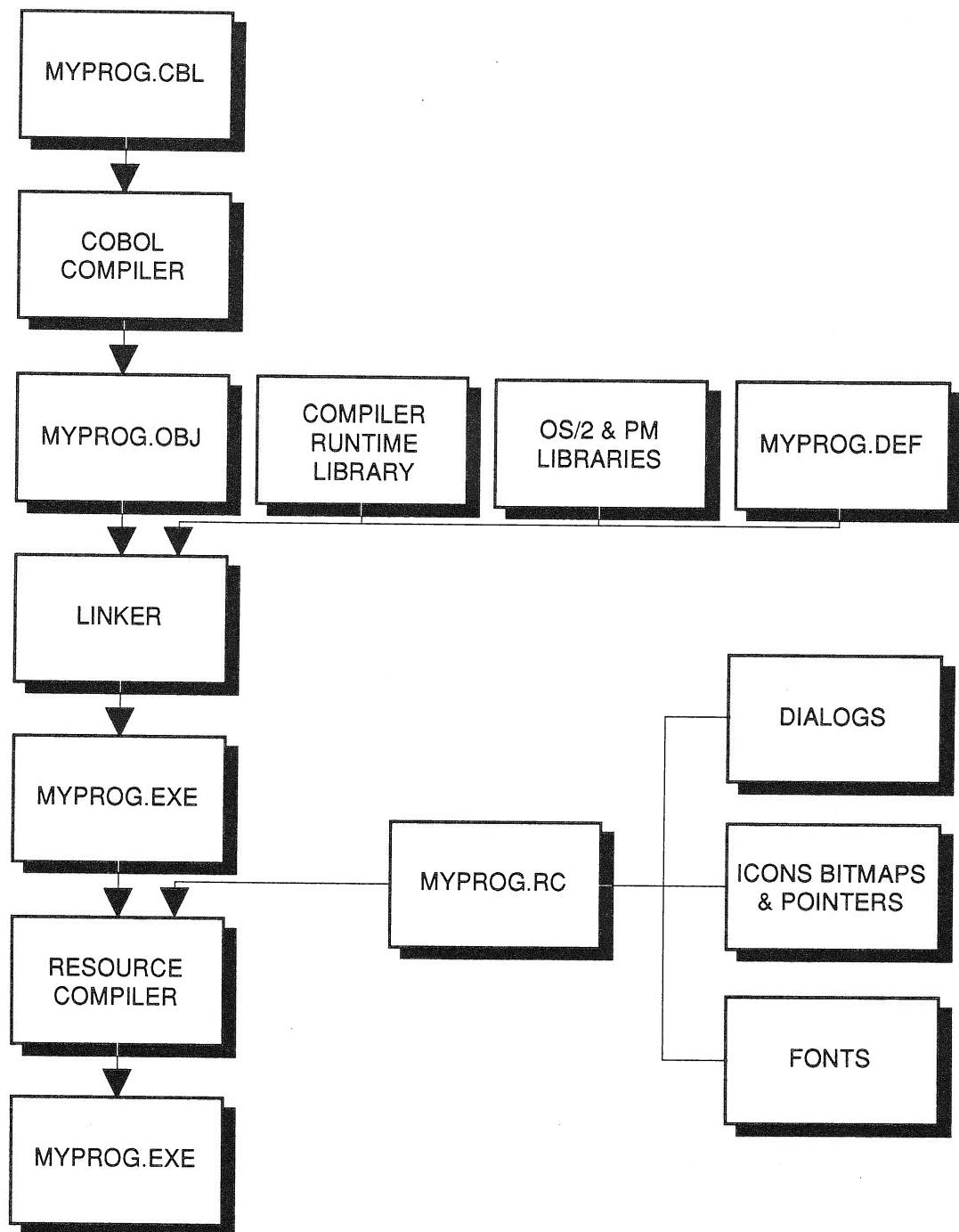


Figure 3-1 The compile, link and bind process for a PM program

```
@echo off
if not exist %1.cbl goto error
Rem *** Compile and Link Program ***
Rem For Xilerator add /GANIM
COBOL %1.CBL /CONFIRM /MF"6" /LITLINK;
If errorlevel 1 goto end
LINK @%1.L
If errorlevel 1 goto end
RC %1.RES %1.EXE
goto end
:error
echo MAKE requires a valid .cbl file
:end
```

A couple of notes are in order here about the MAKE.CMD file.

- Use the OS/2 Linker, not the Linker that comes with your compiler. There have been problems using the link editors supplied with the compiler, when that linker gets out of sync with OS/2 releases due to different release dates and maintenance procedures. An OS/2 maintenance release can affect the linker by making changes to OS/2. Without a new linker, you could have problems with the link step. As a result, I strongly suggest that you use the OS/2 Linker instead of the linker supplied as part of your COBOL compiler product.
- The *rc* line in the command file assumes the use of a resource. Not all PM programs use resources and if yours doesn't, simply comment out, or remove the *RC* line and the following error test. If you are coding the sample program, chapter by chapter, you will not need these statements until Chapter 5.
- The "@" on the Linker command line indicates the use of a Linker Response file in place of the command line entries. See the following section on the Linker Response file.
- Users of the Micro Focus COBOL/2 Workbench's Check and Compile functions do not need the compiler command file.

COBOL Compiler Directives

Compiler directives are used to specify options that affect how the compiler behaves and the type of code that is ultimately produced. With most COBOL compilers, directives can

be placed on the compiler's command line, in a separate file, as part of the source code using a directives statement or a combination of all three. Usually, the installation standards are placed in the directives file, programmer or team directives placed on the compiler's command line and directives unique to a single program placed directly in the source file.

The directives you will use are obviously based upon your COBOL compiler. If you are not using the Micro Focus COBOL/2 compiler, skip this section and use the directives specified by your compiler. For a complete listing of the Micro Focus COBOL/2 directives, see Appendix B of the *Micro Focus COBOL/2 Operating Guide*.

I use a combination of a directives file and command line directives. I prefer placing most of my directives on the command line because command line directives are displayed on the screen as part of each compile when the /CONFIRM directive is specified. Directives taken from the COBOL.DIR file are not displayed. Here are the Micro Focus COBOL/2 compiler directives that I use. The directives marked with an asterisk must be used when compiling PM programs. Other directives may, of course, be added.

/ANS85 - Use the ANS COBOL 85 reserved words, file handling and status words.*

/CONFIRM - Echo all subsequent directives to the screen. This directive should always be listed first.

/GANIM - Tells the compiler to produce the additional files required to run the Xilerator debugger program

/LITLINK - Causes the compiler to declare all literals used in CALL statements as public symbols. This parameter, or the use of the double underscores with the call, is required to generate the external symbol table for the link step.*

/MF"6" - An optional parameter defining the correct level of compatibility with past levels of the Micro Focus compiler. For PM programs, MF"5" or MF"6" must be specified. If /MF is not specified, or /MF without a value is specified, 6 is assumed.

/NESTCALL - Allows nested calls to appear in the program.

The Linker Response File

The Link Response file allows the search library list and Linker directives to be removed from the Linker command line, giving you more freedom to apply different directives and

libraries to each program and automatically associate the correct library list and directives with each COBOL program. Simply give the Linker Response file the same name as the associated COBOL source file, and the Linker will automatically apply these responses during the Linker run.

The Linker Response file contains the same parameters as are coded on the Linker command line or entered in response to the Linker prompt messages. There are six parameters and options required, and the Linker Response file must account for all six. The six parameters are:

OBJLIST - A list of object files that are to be linked together. Each file name should be separated by a plus (+) sign. If the file name extensions are not specified, .OBJ is assumed.

RUNFILE - This is the name of the executable file that the Linker is to create. If you do not specify a name, the Linker uses the name of the first object module from the objlist line. Programs are given the extension of .EXE and dynamic link libraries the extension of .DLL.

LISTFILE - The name of the file that will contain the Link Map listing. The default file extension is .MAP.

LIBLIST - The list of libraries to be searched by the Linker. When more than one library name is specified, each library must be separated by a plus (+) sign.

DEFINITIONSFILE - The name of the Module Definition file for the executable or dynamic link library being created.

/OPTIONSFILE - A list of Linker directives or options.

Here are the rules for building a Linker Response file.

- Each Linker response must be placed on a separate line.
- Where responses require more than one line, a plus (+) sign must be placed at the end each line that is to be continued on the following line.
- A response, regardless of the number of lines used, may not exceed 128 characters in total length.
- A semicolon (;) signals the Linker to supply default responses for all remaining entries and is usually placed at the end of the response file.

- Each Linker option must begin with a "/" character. Options may be grouped on one line or separated on several lines. Options may appear anywhere in the Linker Response file.

Regardless of whether you use a Linker Response File or code the Linker directives on the command line, you will need these parameters as a minimum. Note, MYPROG should be replaced with the name of your COBOL source program.

OS/2 Version 1.3

```
myprog.obj
myprog.exe
nul.map
lcobol.lib+
os2.lib+
pmcob.lib+
pmbind.lib
myprog.def
/NOD /NOP
;
```

OS/2 Version 2.0

```
myprog.obj
myprog.exe
nul.map
lcobol.lib+
os2286.lib
myprog.def
/NOD /NOP
;
```

As with the compile command file, this Linker response file example assumes the use of the Micro Focus COBOL/2 compiler. If you use another COBOL compiler, you will need to substitute that compiler's static run-time library for Micro Focus's LCOBOL static run-time library.

The Module Definition File

The Module Definition (.DEF) file provides the Linker with additional information about the type of executable module to create. Although this is an optional file, you should consider the Module Definition file a requirement for every PM program that you develop. There are several Linker specifications that can only be passed using the Module Definition file. While not every program requires these specifications, by creating a Module Definition file for every program the correct vehicle will be available when these specifications are required.

The Module Definition file specifies the name and description of the executable, its executing environment, its code and data segment attributes, a list of any imported or

exported functions, and the stack and heap sizes for the executable. Here are the Module Definition file entries that you will need for the sample programs. Additional entries may be required, depending upon your environment. For a complete list of module definitions, see the *Presentation Manager Programming References*, part of the Developer's Toolkit.

NAME - Indicates that the file being created is a program (.exe) module. The LIBRARY statement is used to indicate the creation of a dynamic link library module. The Name and Library statements are mutually exclusive.

WINDOWAPI - Indicates that this program uses the APIs provided by the Presentation Manager and must be executed in a Presentation Manager window

Description - Defines text to be inserted at the beginning of the .EXE module. This text can be used for source control or copyright information

DATA MULTIPLE - Defines the default attribute for the application's data segments. MULTIPLE causes a new data segment to be copied for each instance of the module.

STACKSIZE - Defines the local stack size. Initially, the sample program requires a stack of 16,384 bytes. As the sample program grows in complexity, the stack size will need to be increased. The completed sample program requires a stack size of 40,960 bytes.

HEAPSIZE - Defines the local heap as 16384 bytes.

PROTMODE - Sets this as an OS/2 protect mode ONLY program.

Here is the Module Definition file used to build the early versions of the sample program. Note that comments must be proceeded by a semicolon (;).

```
;Sample Module Definition File
name sample windowapi
description 'OS/2 PM COBOL/2 Sample Program. Copyright by
David Dill, 1992'
data multiple
stacksize    16384
heapsize     16384
protmode
```

The Resource File

I will discuss resources in more detail in Chapter 5, when a resource file is added to the sample program. For now, though, a brief word about the Resource Script file is in order.

A resource file is a collection of ASCII text strings managed by OS/2 in read-only data segments for use by programs. Resources are not part of a program's data segment and must be moved to a variable within the program's data segment before they can be used. Placing the read-only text strings in a separate data segment introduces significant flexibility into how these strings are utilized. They may be shared among multiple programs (common error messages, for example), their loading may be deferred until needed by a program (messages for little used subroutines, as one example) and they may be modified without recompiling the programs that use them (for example, change the wording of a message).

The source form of a resource file is called the Resource Script file. It is composed solely of read-only character-based data strings and graphical images of the following types.

- Bit maps
- Dialog and window templates
- Fonts, Icons and Screen pointers
- Keyboard accelerator tables
- Menu tables
- String tables
- User-defined resources

Resource objects (bit maps, dialog and window templates, fonts, icons and pointers) are built as separate files using one of the toolkit editors, the Icon editor, the Font editor or the Dialog Box editor. Once built, references to these resource objects, or the objects themselves, are placed into the Resource Script file. Whichever method is used, these objects are then combined with the ASCII string tables (accelerator, menu or text string) built using any ASCII editor to complete the Resource Script file.

The Resource Script file then undergoes a stage 1 compile using the Resource compiler. This converts the ASCII Resource Script file into a binary file called the Resource file. The binary version of the Resource file is then either bound with the executable program or into a separate dynamic link library using stage 2 of the Resource compiler. Stages 1 and 2 of the Resource compiler do not need to be run back-to-back. Resource files that are bound with the executable are usually kept in their binary form and simply bound with the executable every time the executable is recompiled. Resource files kept as separate dynamic link libraries are usually stored in their finished form.

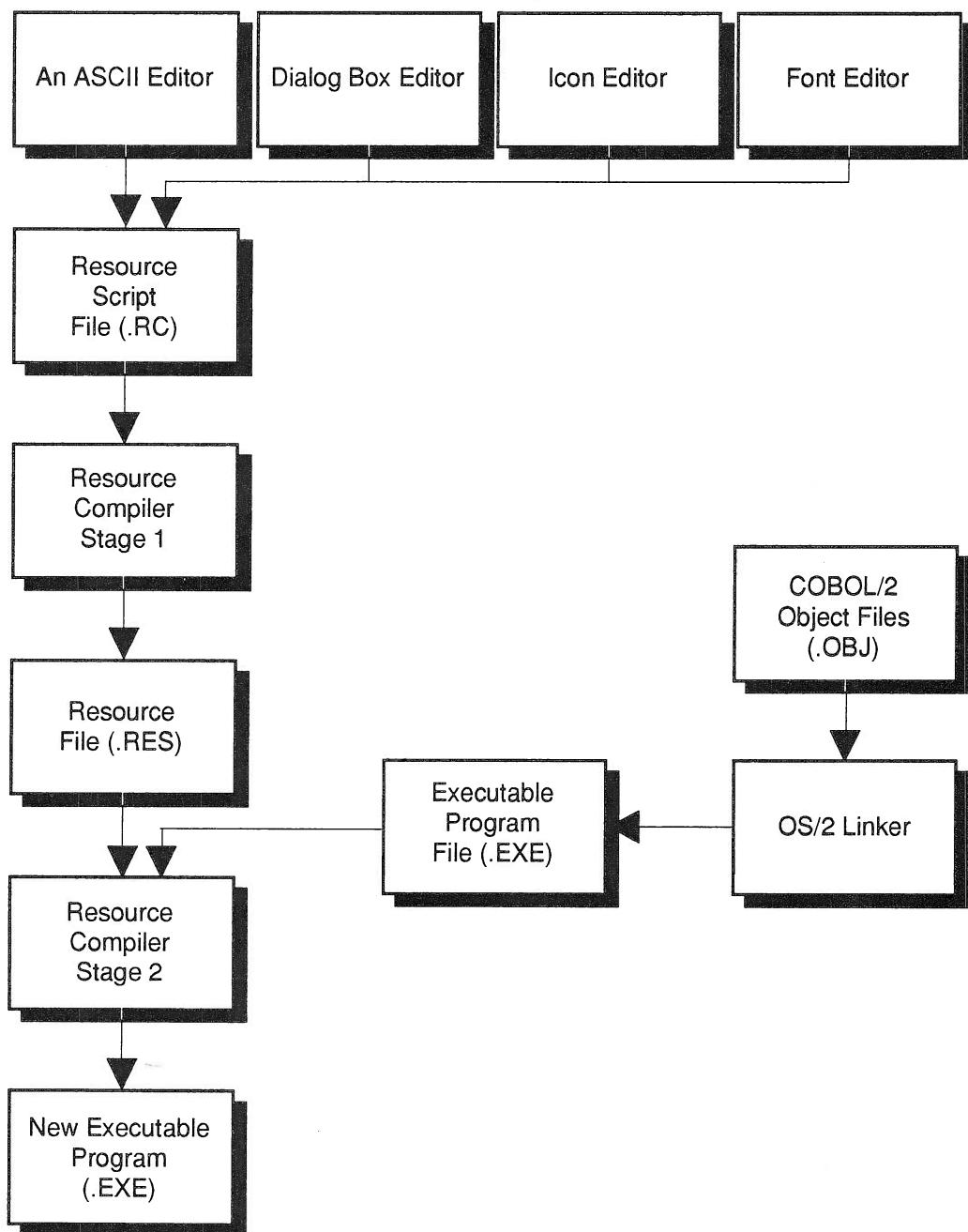


Figure 3-2 The resource compile process

As read-only files, resources may not be altered except by changing the Resource Script file and going back through the compilation and binding process. But, changing a resource usually does not require changing, recompiling or relinking the program that uses it.

The Resource Compiler Command File

The Resource Compiler may be run as separate compile and bind processes, or with the compile and bind process run back-to-back. For the sample programs, I use separate compile and bind processes. The Resource Script file must be compiled only when it is changed. This is rather infrequent when compared to the number of COBOL compiles that are run, especially during program development. By separating the resource compile from the resource bind and including only the bind with the COBOL compiles, significant time is saved with every COBOL compile.

To compile the Resource Script file you will need a command file similar to the command file used to compile your COBOL programs. This file contains a single line entry required to execute the Resource compiler. The -r switch instructs the Resource Compiler to store the compiled stage 1 output as a file. The compiled resource is given the same name as the Resource Script file but with a file extension of .res.

When separate resource compile and binds are used, the MAKE.CMD file must execute the Resource compiler to bind the binary resource file with an executable program or dynamic link library. To bind the binary resource file to the application or .DLL, insert a single line into the MAKE.CMD file to execute the Resource compiler. Place the name of the resource file (.RES) as the first parameter, followed by the name of the linked application module (.EXE or DLL) as the second parameter. The %1 characters used in the MAKE.CMD file, allows the name entered by the user as the first parameter of the MAKE command, to be inserted into the resource compile statement as the file name.

The resource compile process can be combined with the bind process to make the resource file changes immediately available to your program. To immediately implement changes to the Resource Script file, I use the BUILD.CMD command file shown below. The first line of this file executes stage 1 of the Resource compiler to convert the Resource Script file to a binary form. The second line executes stage 2 of the Resource compiler to bind the resource file to the sample program. As before, the %1 characters allow the file name to be inserted at execution time. Note that the stage 2 bind statement in this command file is identical to the stage 2 bind statement in the Make command file.

```
@echo off
if not exist %1.* goto error
REM **** BUILD THE RESOURCE .RES FILE ****
REM **** Line 1 compiles the resource file ****
REM **** Line 2 binds the resource to the program ****
rc -r %1.rc
rc %1.res %1.exe
goto end
:error
echo BUILD requires a valid .rc file
:end
```

The Icon Editor

The Icon editor is one of the special editors supplied as part of the Developer's Toolkit. The Icon editor is used to create bit maps, one of the resource objects that are input to the Resource compiler. These bit maps may be used as bit maps, icons or screen pointers.

A complete description of how to use the Icon editor is contained in the *Presentation Manager Programming References* supplied as part of the Developer's Toolkit. I have not included instructions on the use of this editor; the manual gives you the necessary detailed information. When using the editor, however, you should understand the display screen environment that will be used to run your programs.

The editor offers three basic drawing densities: 64 x 64 pels for XGA displays, 40 x 40 pels for 8514/A displays, 32 x 32 pels for VGA or EGA displays and 32 x 16 pels for CGA displays.

The higher pel density allows for better drawing, but there are problems when these bit-mapped objects are drawn at higher densities than will be used for their display. 64 x 64 pel bit maps displayed at the lower 40 x 40 pel or 32 x 32 pel density will appear in an degraded mode, drawn using only selected rows of the original 64 x 64 bit map. These anomalies can significantly alter the appearance of a bit map, icon or pointer. So, unless you know the exact target machines for your program, I suggest that you use the 32 x 32 pel drawing option.

The normal installation process installs this editor into the \TOOLKT13\BIN subdirectory for Version 1.3 or the \TOOLKT20\OS2BIN subdirectory for Version 2.0. To execute the Icon editor, select *Toolkit Editors*, then the *Icon Editor* from the Toolkit Editors menu or folder.

The Dialog Box Editor

The Dialog Box editor is another of the special editors supplied as part of the Developer's Toolkit. The Dialog Box editor is used to create the templates for pop-up dialog boxes, windows and one or more of the following types of control windows.

- Check boxes
- Combo boxes
- Containers
- Frames windows
- Group boxes
- Horizontal and vertical scroll bars
- List boxes
- Multiple list entry boxes
- Notebooks
- Push buttons
- Radio buttons
- Sliders
- Spin buttons
- Text entries
- User defined controls
- Value sets

The Dialog Box editor does not create the actual control, but rather a series of ASCII drawing instructions, called a template, that PM uses to draw the control or window when the program issues the command to create the box or window. The output of the Dialog Box editor may be stored as a separate file and referenced from within the Resource Script file using the RCINCLUDE statement, or the output may be inserted directly into the Resource Script file. Either way produces the same results, but the RCINCLUDE statement gives a cleaner Resource Script file and more flexibility in relating standard windows and controls to custom resource scripts.

The normal installation process installs the editor into the \TOOLKT13\BIN subdirectory for Version 1.3 or the \TOOLKT20\OS2BIN subdirectory for Version 2.0. To execute the Dialog Box editor, select *Toolkit Editors*, then *Dialog Box Editor* from the Toolkit Editors menu or folder.

The Font Editor

The Font editor is the third of the special editors supplied as part of the Developer's Toolkit. The Font editor is used to create custom image fonts, also called raster fonts, for use within Presentation Manager windows.

The Presentation Manager provides several typographic-quality image fonts that are available for all OS/2-supported display adapters. The supplied fonts are Courier, Helv, Times Roman and System Monospaced in a variety of sizes, from 8 through 24 points (Courier is not available in all sizes). You may not edit these fonts, but you may create custom image fonts using the Font editor. Every font that is created requires a Font statement reference to the font file placed in the Resource Script file.

As with the other editors, I have not included instructions on the use of this editor; the manual gives you the necessary detailed information. A complete description of how to use the Font editor is contained in the *Presentation Manager Programming References* supplied as part of the Developer's Toolkit. You will not be using the Font editor with this book.

The normal installation process installs this editor into the \TOOLKT13\BIN subdirectory for Version 1.3 or the \TOOLKT20\OS2BIN subdirectory for Version 2.0. To execute the Font editor, select *Toolkit Editors* then *Font Editor* from the Toolkit Editors menu or folder.

Chapter 4

Coding The Basic PM Program

This chapter establishes the basic structure of a COBOL Presentation Manager program. No matter how your program is designed or its intended function, this structure will be at the heart of every COBOL Presentation Manager program that you write. It is very important that you understand the structure and data flow of this skeleton PM program. If you can master the structure and understand why this code is written the way it is, you will have come a long way toward understanding the basics of Presentation Manager programming.

The skeleton PM program contains the four standard sections found in all PM programs, the initialization routine, the message processing routine, the termination routine and the first window procedure. Whenever possible, the calls used within this program have only the minimum parameters specified. In future chapters, as the calls become more complex, many of these missing parameters will be completed.

The window created by this program is a simple frame window with no embedded control windows and no application window, certainly not a finished or usable window. This window will be enhanced in future chapters. The objective for now is to master the basics of all PM programs and not worry about the quality of the window.

A Word About The Sample Program

I designed the sample program to be modular, allowing each chapter to build upon the work done in the prior chapters, while concentrating on a major Presentation Manager function and isolating the code required to perform it. The completed sample program is not a good example of a Presentation Manager program; it was never intended to be a demonstration program. Rather, it is a working repository of routines that allows you

to see how a function works, then extract the code necessary to perform that function and move it to your application. By the end of the book you will have a collection of code segments, routines and parts of routines that invoke most of the Presentation Manager functions necessary to write a complete PM application.

A word about return code and error processing in the sample program is in order. You will quickly note that the sample program contains no return code or error processing. I thought long and hard about including this code, but eventually decided not to for the following reasons.

- There is nothing unique about Presentation Manager return code and error processing. It is the same as the return code and error processing found in every COBOL program. Most COBOL programmers are very good at this kind of coding.
- The important piece of PM return code and error processing is understanding exactly what information is returned by each call. I have tried to include that information with the explanation of each call and in Appendix B.
- Adding return code processing to the sample program would significantly increase the complexity of the code, making it more difficult to read and harder to understand, while clouding the interrelationship of the PM calls.
- This book is designed as an introduction to PM programming, and learning the unique features of a multithreaded, reentrant, message-based application is easier when only the unique PM calls are included.

This does not mean that PM programs can be written without return code and error processing. Such a program would be worthless. The dynamics of the PM environment make return code and error processing mandatory in every PM program. Please do not confuse the specialized code I have written with what is required for good PM programming.

Now that you have completed the necessary software installations and setup functions described in Chapter 3, you are ready to begin writing a COBOL Presentation Manager program. Unfortunately, as exciting as this might sound, you will need to write a good deal of standard COBOL code before the windows begin to appear. Despite the fact that these programs run under the control of the Presentation Manager, they are still COBOL programs, and they still need an Identification Division, an Environment Division, a Working-Storage Section, a Procedure Division and all the other standard COBOL sections.

As you proceed through the sample program, I will discuss each new Presentation Manager call in detail, but I will not spend time on code that is part of every COBOL program. I have included only the minimum standard COBOL code required to compile

and run the sample program. Limiting the standard COBOL code makes it easier to see the PM code, how the PM calls are written and how they relate to each other. And that, after all, is the objective of this book. It was always my intention that you would customize my code with additional COBOL code.

The complete source code for the finished sample program is contained in Appendix A. To help you understand the new code required for each iteration of the program, I have included the source line numbers with each reference to specific PM calls. However, the program introduced in this chapter should be used as your basic PM starter program, so I have included its source code at the end of this chapter. For this chapter only, the source numbers used in the examples refer to the code at the end of this chapter, not the source code in Appendix A.

The Identification Division

The use of the Presentation Manager with a COBOL program does not change the basic structure of the COBOL program and the standard Identification Division statements are still required. Since there is no special coding required for this division, it should look familiar. The following Identification Division is used in the sample program.

```
000010 IDENTIFICATION DIVISION.  
000020 PROGRAM-ID. COBOL-PM-SAMPLE.  
000030 DATE-COMPILED. 01-02-92.  
000040 AUTHOR. DAVID DILL.
```

The Environment Division

As with the Identification Division, there are no special requirements for the Environment Division within a COBOL Presentation Manager program. The following Environment Division was used for the sample program.

```
000060 ENVIRONMENT DIVISION.  
000070 CONFIGURATION SECTION.  
000080 SOURCE-COMPUTER. IBM-PERSONAL-SYSTEM-2.  
000090 OBJECT-COMPUTER. IBM-PERSONAL-SYSTEM-2.
```

The Working-Storage Section

The Working-Storage Section is a combination of data definitions that must appear in every PM program together with definitions that are unique to each program. Since this chapter is concentrating on the basics, I will take some time to discuss those entries required in every Presentation Manager program.

First to be defined is the structure of the Presentation Manager message. This structure must appear in every PM program. Every message sent from or to PM must adhere to this structure. The structure is defined as follows.

QMSG-HWND: An unsigned long integer containing the handle of the message's target window. PM uses this handle to route messages to the correct window procedure.

QMSG-MSGID: An unsigned short integer containing the message identification number. This field defines the message. See Appendix C for a list of the most common PM messages and their IDs.

QMSG-PARAMS: Two unsigned long integers containing the data passed by the message. The content and structure of the data within these parameters depend upon the specific message. See the *Presentation Manager Programming References* for detailed information on the content and data structure of the message parameters for each PM message.

QMSG-TIME: An unsigned long integer containing the time that the message was generated.

QMSG-POINT: Two unsigned long integers that contain the x and y coordinates of the mouse pointer at the time the message was generated.

Here is the standard message structure:

```

000180 01 QMSG.
000190      05 QMSG-HWND          pic 9(9) comp-5.
000200      05 QMSG-MSGID         pic 9(4) comp-5.
000210      05 QMSG-PARAM1        pic 9(9) comp-5.
000220      05 QMSG-PARAM2        pic 9(9) comp-5.
000230      05 QMSG-TIME          pic 9(9) comp-5.
000240      05 QMSG-POINT.        pic 9(9) comp-5.
000250      10 QMSG-X            pic 9(9) comp-5.
000260      10 QMSG-Y            pic 9(9) comp-5.

```

Every PM program requires a minimum of two handles, the Anchor-Block handle, identifying the program to PM and the Message Queue handle, identifying the program's message queue to PM. If the program creates a window, then one or two additional handles are required, depending upon the type of window created.

When a single window is created using the WinCreateWindow call, the handle of that window is returned. When a standard window is created using the WinCreateStdWindow, two handles are returned. A standard window is a frame window containing one or more control windows and an application, or client, window (see Figure 4-1). The WinCreateStdWindow returns the handles of the frame window and the client window. Clearly, your PM applications will have more than one window and many more handles than the four in this basic program, but every program that creates a window must have these four handle variables defined, as a minimum.

The sample program includes two additional handles (statements 340 and 350) that point out additional uses of handles. Handles are not used just for creating application windows. Handles also identify standard elements of OS/2 and PM. The handle of the desktop (HWND-DESKTOP) identifies the display screen that forms the background of the PM session. The desktop is owned by PM and always present within the PM session. As such, it has the predefined handle of 1. The HWND-TOP illustrates another use for handles, a shorthand notation for window manipulations. HWND-TOP specifies the highest Z-order position among sibling windows. Similarly, HWND-BOTTOM is the predefined specification for the lowest z-order position among sibling windows.

000300 77 hab	pic s9(9) comp-5.
000310 77 hmq	pic s9(9) comp-5.
000320 77 hwndClient	pic s9(9) comp-5.
000330 77 hwndFrame	pic s9(9) comp-5.
000340 77 HWND-DESKTOP	pic s9(9) comp-5 value 1.
000350 77 HWND-TOP	pic s9(9) comp-5 value 3.

Window class names identify window classes. A window class is the unique combination of window styles, control window content and code procedure that makes the windows of one class unique from windows of another class. A unique name is assigned to each class at class registration time and may be any user-defined null-terminated character string. But, the class name should have some relevance to the window class that it defines.

The sample program's class names are an example of how to code a static null-terminated string. Remember, PM has no notion of character variables, so every string passed to PM must have a null character indicating the end of the string. The length value of the string must include the appended null character.

```
000390 77 MainWndClass  pic x(9) value "MainClas" & x"00".
```

The class style word, window style word, frame control word and window position word are the binary sum of the individual flags selected for each window. Each flag adds a unique style, function, control window or positioning element to the frame window. When creating a standard window, you must include sufficient frame control flags to define the content and manipulation characteristics of the frame window. A discussion of the class style, window styles and frame control flags, along with a description of each flag is contained in Chapter 1.

Window position flags define how the window is to be displayed, its size, z-order, its current status, and whether it is to be hidden or visible. A window's position flags are set exactly like the other style flags, by creating an integer value that is the binary sum of the flags to be used. Here is a list of the individual set window position flags along with a description of each flag.

SET WINDOW POSITION FLAGS

SWP-ACTIVATE	Activate the designated window.
SWP-DEACTIVATE	Deactivate the designated window.
SWP-HIDE	Hide the designated window.
SWP-MAXIMIZE	Fill the desktop with the designated window.
SWP-MINIMIZE	Remove the designated window from the desktop and represent it as an icon at the bottom of the desktop.
SWP-MOVE	Change the designated window's x, y coordinates.
SWP-NOADJUST	Do not allow adjustment of the designated window before moving or sizing it.
SWP-NOREDRAW	Do not redraw the designated window to reflect changes.
SWP-RESTORE	Restore the designated window from icon status to a window on the desktop with the size and position specified.
SWP-SHOW	Make the designated window visible to the user.
SWP-SIZE	Change the designated window's cx, cy coordinates.
SWP-ZORDER	Change the designated window's relative placement among its sibling windows.

Here are the class style, frame control window and set window position words used in the sample program. You may wish to modify these flags to better meet your requirements.

```

000440 77 CSCLASS      pic 9(9) comp-5 value h"20000000"
000550 77 FCF-MAIN    pic 9(9) comp-5 value h"00000a3b"
000640 77 SWP-WINDOW   pic 9(9) comp-5 value h"0000008f"

```

Regardless of a program's design, every program must check for and process the WM-QUIT message. The WM-QUIT message is sent to a program whenever the user elects to end the program by selecting a program termination entry from a program menu, system menu, the task list or the system shutdown entry. In this chapter's version of the sample program, the WM-QUIT message is generated when the user selects *Close* from the System Menu. In later chapters, I will add additional ways for the user to indicate that the program should be ended.

The WM-QUIT message identifier, like all message identifiers, is defined as a standard signed, short integer. A list of the most common PM messages is included as Appendix C. Here is the WM-QUIT message declaration in the sample program:

```

00730 77 WM-QUIT      pic s9(4) comp-5 value 42.

```

The Local-Storage Section

As discussed in Chapter 2, recursion is an important feature of all PM programs. To help with the problems of recursion, you must make extensive use of local storage within your programs. As with all COBOL programs, defining variables in local storage ensures a new allocation of the variable for each recursion. As a general rule, all variables set or updated by a window procedure must be defined in local storage. This results in COBOL PM programs having large Local-Storage sections. The size of the Local-Storage section in the final version of the sample program is the main reason that its stack requires in excess of 40KB bytes.

A word of warning here. As the local storage in your programs grows, pay special attention to the size of your program's stack. Local storage is allocated from the stack and has a big impact on the amount of stack space required. Prior to execution, there is no indication that your program's stack is too small, at the moment the stack overflows the program is terminated.

There are no local storage variables required in every COBOL Presentation Manager program. The contents of local storage depends upon how the window procedures are coded, but here are the items required by the skeleton sample program. Most of these variables are a safe bet to wind up in your PM programs. The *Mresult* variable holds the

value returned by each procedure. The two size variables are required by several PM system measurement calls and the window size data structure (rcl) is required when moving or positioning a window.

```
000910 LOCAL-STORAGE SECTION.  
000920 01 Mresult      pic x(4) comp-5.  
000930 01 SizeWide    pic s9(9) comp-5.  
000940 01 SizeTall    pic s9(9) comp-5.  
000950 01 rcl.  
000960 05 XLeft       pic s9(4) comp-5.  
000970 05 filler      pic s9(4) comp-5.  
000980 05 YBottom     pic s9(4) comp-5.  
000990 05 filler      pic s9(4) comp-5.  
001000 05 XRight      pic s9(4) comp-5.  
001010 05 filler      pic s9(4) comp-5.  
001020 05 YTop        pic s9(4) comp-5.  
001030 05 filler      pic s9(4) comp-5.
```

The Linkage Section

Remember, PM programs are nothing more than a collection of independent procedures with data continually flowing among them. The Linkage Section supports this data passing function by defining the form and content of data that is passed from procedure to procedure. Since PM messages represent the majority of data passed between procedures, the standard message structure must be defined in the Linkage Section of every PM program.

A close look will show that the Linkage Section message structure is not the same as the message structure defined in the Working-Storage Section. In the Linkage Section the two message parameters are redefined so that each can be referenced as a single long integer or two short integers. The Main Section of your program never deals with the specifics of a message so the contents of the message parameters are unimportant. Window procedures, however, deal extensively with message parameters so the parameters must be redefined to account for every possible data structure. Here is the PM message definition used in the Linkage Section of every PM program:

```

001050 LINKAGE SECTION.
001060 01 hwnd           pic 9(9) comp-5.
001070 01 Msg            pic 9(4) comp-5.
001080 01 MsgParm1       pic 9(9) comp-5.
001090 01 Redefines MsgParm1.
001100    05 MsgParm1w1   pic 9(4) comp-5.
001110    05 MsgParm1w2   pic 9(4) comp-5.
001120 01 MsgParm2       pic 9(9) comp-5.
001130 01 Redefines MsgParm2.
001140    05 MsgParm2w1   pic 9(4) comp-5.
001150    05 MsgParm2w2   pic 9(4) comp-5.
001160 01 MsgTime        pic 9(9) comp-5.
001170 01 MsgXPointer    pic 9(9) comp-5.
001180 01 MsgYPointer    pic 9(9) comp-5.

```

The Procedure Division

Now that you have completed the required Working-Storage, Local-Storage and Linkage Section declarations, you are ready to code the procedure division of your program and begin invoking Presentation Manager services. The following parts of the Main Section of every PM program are identical. They are always coded the same and never vary in their order of execution. Obviously, additional calls are usually present within the Main Section, as you will see in later chapters, but you will always find these sections and always in this order.

The Main Section of every COBOL Presentation Manager program can be broken down into the following three subsections.

1) The Initialization Routine

Initialize the Presentation Manager interface, using the WinInitialize call. Create the message queue for your program, using the WinCreateMsgQueue call.

Register the class name, class style and procedure address of each private window class that will be used by the program, using the WinRegisterClass call.

Create the Main Window, using the WinCreateWindow or WinCreateStdWindow call.

2) The Message Processing Routine

Get the next message from the message queue using the WinGetMsg call.

Check for the WM-QUIT message indicating the user wishes to end the program. If WM-QUIT is received, execute the program's termination routine.

Dispatch all messages to the correct procedure using the WinDispatchMsg call.

Return to the WinGetMsg call and wait for the next message.

3) The Termination Routine

Destroy any remaining windows, using the WinDestroyWindow call.

Destroy the message queue, using the WinDestroyMsgQueue call.

Terminate the Presentation Manager interface using the WinTerminate call.

Coding The WinInitialize Call

The first call to the Presentation Manager in every program must be the initialization of your program's Presentation Manager interface and the return of the Anchor-block handle from PM. A significant number of resources and control blocks must be established before a program can invoke any PM services. This initialization call causes PM to establish these resources. Parameter one is not used with OS/2's PM so it must always be set to null. PM returns the Anchor-block handle at the conclusion of the call in the signed, long integer variable coded as parameter two. Remember, the second parameter will be updated by PM, so a pointer to the *hab* variable must be passed to PM not the actual variable. To specify a variable pointer rather than the actual variable, the *by reference* phrase must be included. Here is the WinInitialize call:

```
001380      Call OS2API 'WinInitialize' using
001390              by value      ShortNull
001400              by reference hab.
```

Coding The WinCreateMsgQueue Call

Now that the program's PM interface has been initialized, the program must request the establishment of a message queue to begin receiving PM messages. Remember, the program is subject to receiving messages at any time after initialization. To avoid losing

messages, the program must establish a queue immediately after the WinInitialize call, using the WinCreateMsgQueue call.

To request a message queue for your program, issue the WinCreateMsgQueue call. The first parameter is the variable holding the Anchor-block handle, obtained during the prior WinInitialize call. The second parameter is used to specify the number of entries in the message queue. Parameter two should be specified as a null to accept the system default message queue size. Unless your program is extremely slow processing messages, there is no need to increase the number of entries in the standard message queue. When this call completes, the Message Queue handle, returned by PM as the call's return code, is placed into the signed, long integer variable specified as parameter three. Here is the WinCreateMsgQueue call:

```

001440      Call OS2API 'WinCreateMsgQueue' using
001450                  by value hab
001460                  by value ShortNull
001470                  returning hmq.

```

Coding The WinRegisterClass Call

Before a window using a private class can be created, that class must be registered with PM. Registration of a window class passes the address of the supporting code procedure and class window styles to PM, allowing PM to associate these values with all windows within the class, under the class name. Remember, however, you are registering window classes, not individual windows. There may be more than one window using the same class, requiring only a single registration call. While you can register a window's class any time prior to creating the window, PM coding conventions recommend that all the window classes be registered at this point.

Prior to registering the window class, the address of the supporting window procedure's entry point must be established so that it can be passed to PM during registration. To establish the address of the procedure's entry point, use the COBOL *Set* statement to load the address of the procedure name, specified in the entry statement, into a variable declared as a procedure-pointer. This variable is then coded in the WinRegisterClass call, effectively passing the address of the window procedure's entry point to PM. Note that the address is passed using the *by value* phrase as the procedure-pointer contains the address. If the *by reference* phrase were used, a pointer to the procedure-pointer variable would be passed, rather than the contents of the procedure-pointer variable.

To register your window classes, issue the WinRegisterClass call once for each class to be registered. Code parameter one with the variable holding the Anchor-block handle.

The second parameter appears to be the variable holding the class name, but the by reference phrase means that this parameter contains a pointer to the first character in the string, not the entire string. Remember, this string must include the null-terminator character. Parameter three is the variable holding the address of the window procedure to be associated with all windows using this class. All messages for this window will be dispatched to this address. To set the window's class style, place the name of the variable that holds the Class Style flags in the fourth parameter. The fifth parameter is the number of extra bytes to set aside for each instance of this window. This variable, called Window Words, will be discussed in Chapter 16, so for now request no extra storage by coding this parameter as a null. You may wish to change this and make some amount of window words standard in all your programs. If so, see Chapter 16 for information on implementing window words. The return code from the call is placed into the variable coded as parameter six. Here is the WinRegisterClass call used in the sample program:

```
001510      set WindowProc to ENTRY 'MainWndProc'
001520      Call OS2API 'WinRegisterClass' using
001530          by value      hab
001540          by reference MainWndClass
001550          by value      WindowProc
001560          by value      CSClass
001570          by value      ShortNull
001580      returning     ReturnData
```

Coding The WinCreateStdWindow Call

It is now time to request construction of the sample program's frame window. The selected style flags and frame control flags will determine the exact appearance of the window.

As you will see when the window is displayed, the application window, also referred to as the client window, appears hollow with the desktop showing through the hole where this window should be. In Chapter 6, I will discuss how to fill this hole in the window, but for now, this window is an excellent example of exactly how much of a window's function is actually provided by the frame window. By manipulating the completed window you will notice that the frame window provides the moving, sizing, maximizing and minimizing capabilities of the window and all of the controls within the window, assuming that you have specified these functions with the FCF-flags. What this means, of course, is that the default window procedure for the frame window contains the code necessary to perform these tasks.

To request PM to create a window you can use either the WinCreateWindow or WinCreateStdWindow call. Until you become more familiar with PM programming use the WinCreateStdWindow call, because complex windows are easier to build using this single call than a series of WinCreateWindow calls.

The first parameter of the WinCreateStdWindow call is the handle of this window's parent window. Since this is the program's first window, there is no parent window within the program. Yet, every window must have a parent window so the desktop is used as the parent of a program's first window. Remember, no window may exist outside of its parent's frame window. By specifying the desktop as the parent of a window, the clipping and movement limits of that window are set equal to the desktop, allowing the window to be moved anywhere on the desktop and sized up to the size of the desktop. So, place the handle of the desktop as parameter one to make the desktop the parent and owner of the program's Main Window.

The second parameter passes the window style flags. For this frame window no window style flags beyond the class style flags are required, so this parameter is coded as a null value. Parameter three passes the Frame Create flags, those flags that indicate which control windows are to be included within the frame window and what functions the default frame window procedure must perform (moving, sizing, minimizing, etc.). A pointer to the variable holding the FCF- value is actually passed with the call, not the variable itself, as indicated by the *by reference* phrase. Parameter four is the window class name and as with all strings, a pointer to the class name is actually passed.

The next four parameters will not be used until later chapters, so for now indicate that they are to be ignored by entering a null string and null values for parameters five, six, seven and eight. I will discuss the purpose of these parameters as they are added to the sample program.

The final two parameters are the variables that will receive the window handles returned by this call. Parameter nine will receive the handle of the Application, or Client window. While this window appears to be just a hole, it is actually present, just invisible for this iteration of the sample program. Even though it is invisible, it has all of the capabilities of any window. The handle of the frame window is returned as the return code of the WinCreateStdWindow call, so the variable to hold this handle is coded as the last parameter using the *returning* phrase.

Here is the WinCreateStdWindow call for the sample program:

```
001620      Call OS2API'WinCreateStdWindow' using
001630              by value      HWND-DESKTOP
001640              by value      LongNull
001650              by reference FCF-MAIN
001660              by reference MainWndClass
001670              by reference NullString
001680              by value      LongNull
001690              by value      ShortNull
001700              by value      ShortNull
001710              by reference hwndClient
001720          returning      hwndFrame
```

Sizing And Positioning The Window

Because this frame window is so simple, no window style was needed and no style word was passed when the window frame was created. This does not mean that the frame window will have no style; the class style will be used for this window, as with all windows of this class. The flag instructing PM to make the window visible (WS-VISIBLE) is not a class style, so without window styles, PM will create a fully functional but invisible frame window. With few exceptions, every window you create should be created invisible. Creating an invisible window allows your program to make adjustments to the window after it is created but before the user can see or interact with it. Adjustments include sizing and positioning the window, painting the window, displaying initial values within the window and adding or removing control windows based upon current circumstances.

In the sample program the size of the window is adjusted and the window centered on the desktop before it is made visible. To do this, the program queries PM for the size of the desktop, then calculates an area 70 percent of the height and 80 percent of the width of the desktop. The window frame is then centered on the desktop. Finally, the z-order of the window is set to top and it is made visible.

Measuring The Size Of The Desktop

The size of desktop is one of almost fifty system values that can be obtained from PM with the WinQuerySysValue call. The value returned by this call depends upon the code passed to PM as parameter two. The system value codes are listed in Appendix D.

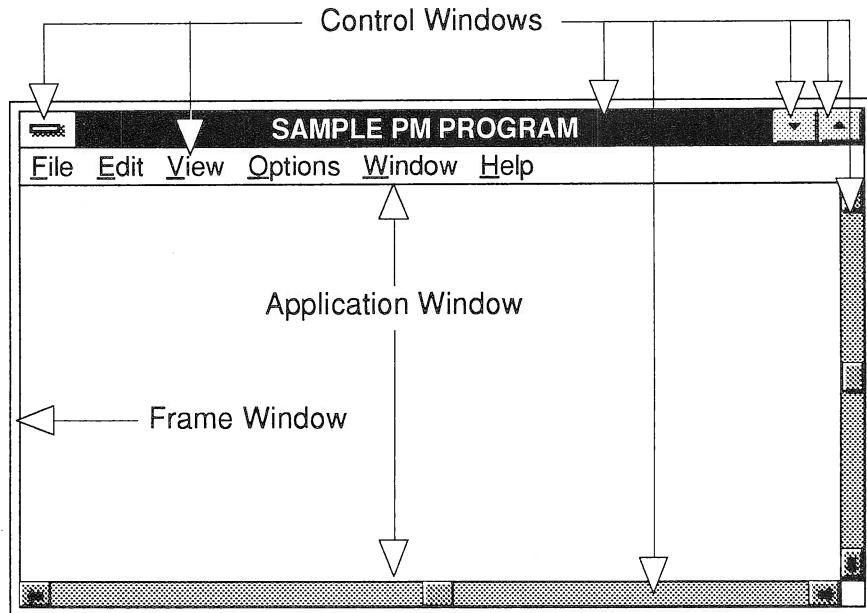


Figure 4-1 The elements of a standard window

To determine the size of the desktop, the WinQuerySysValue call must be issued twice, first to obtain the size of the desktop along the x-axis then to obtain the size along the y-axis. The calls are coded identically with only the SV- code altered to indicate the x-axis, then the y-axis. The handle of the window to be measured is coded as parameter one. For system values the desktop must be queried, so HWND-DESKTOP is coded as parameter one. Parameter two is a short integer containing the system code that identifies the parameter to be returned. Parameter three is the name of the variable that will receive the requested value, returned by PM as the call's return code. Here is the WinQuerySysValue call to measure the width (x-axis) of the desktop:

```

001760      Call OS2API 'WinQuerySysValue' using
001770                  by value HWND-DESKTOP
001780                  by value SV-CXSCREEN
001790                  returning SizeWide

```

Positioning The Window

With the height and width of the desktop available as the variables SizeWide and SizeTall, it is simply a matter of multiplication to set 70 percent of the height and 80

percent of the width.

A Window's location and size on the desktop are determined by four values. The x and y coordinates determine the lower left corner of the window. The x coordinate sets the lower left corner of the window along the desktop's horizontal axis. The y coordinate sets the lower left corner of the window along the desktop's vertical axis.

The cx and cy values determine the size of the window by specifying the height and width of the window. Since the desired height and width of the window are already set, only the x and y coordinates setting the lower left corner need to be calculated. This point is calculated by subtracting the desired height and width of the window from the height and width of the desktop and dividing the result by two. This sets the distance from the edge of the desktop to the edge of the window along the x-axis and y-axis pinpointing the x, y coordinates.

The size of the window (cx and cy) along with the lower left corner of the window (x and y) must be passed to PM to allow the correct sizing and positioning of the window. But, passing these values alone is not sufficient. If the set window position flags do not specify that the window can be sized and positioned (SWP-SIZE and SWP-MOVE), these four values will be ignored. In addition, the SWP-SHOW flag must be set to make the window visible. The sizing, positioning and display of a window is accomplished using the WinSetWindowPosition call.

To code this call, place the variable holding the handle of the window to be positioned in parameter one. When positioning or sizing a window, the window frame is always specified, so for the sample program *hwndFrame* is specified. Parameter two sets the relative placement of the window along the z-axis. A window's position may be on top of(HWND-TOP) or behind (HWND-BOTTOM) all sibling windows, or behind the sibling window whose handle is passed in this parameter. The sample program's window is to be on top of all sibling windows, so the predefined handle, HWND-TOP, is passed as parameter two. Parameters three through six are the four window coordinates, x, y, cx and cy. These values are signed short integers, but the structure that passes them, rcl, must be composed of long integers. To enable the short integer to be correct;y passed, the rcl structure is defined with fillers to correctly position the four short integer values. Parameter seven contains the set window position flags, passed to PM to indicate the allowable window manipulations. For the sample program, the window is to be sized and moved, have the z-order set, be made visible and made the active window. All of this is passed in the variable SWP-WINDOW as the the sum of the individual flags, 0x'8F'. The last parameter is the variable that will receive the return code from the call. Here is the WinSetWindowPos call from the sample program:

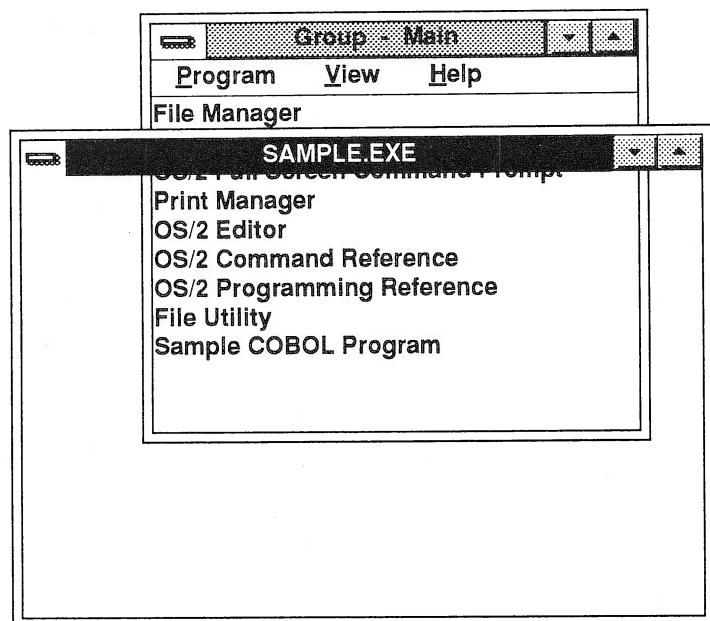


Figure 4-2 The sample program's Main Window (Version 1.3)

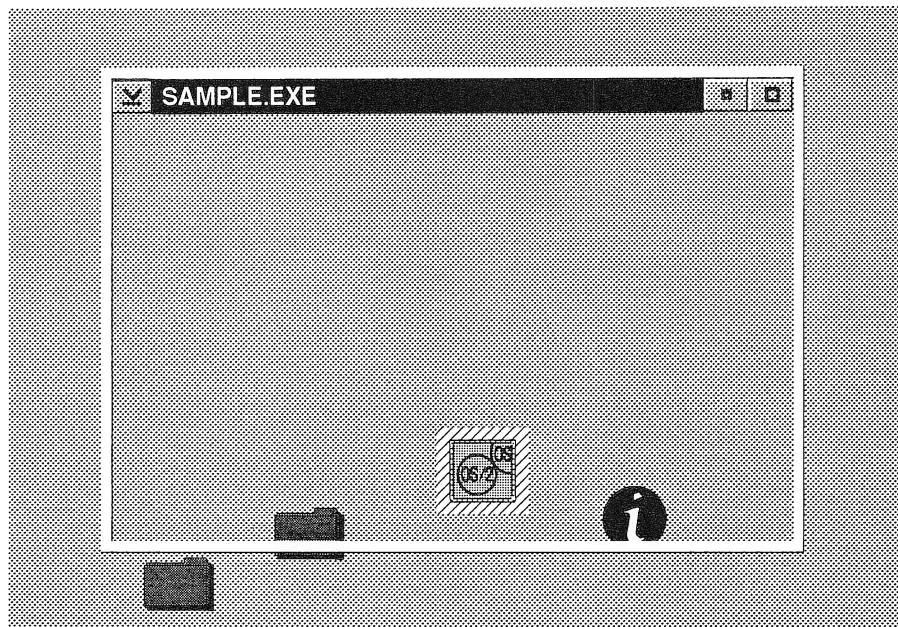


Figure 4-3 The sample program's Main Window (Version 2.0)

```
001960      Call OS2API 'WinSetWindowPos' using
001970              by value hwndFrame
001980              by value Hwnd-TOP
001990              by value XLeft
002000              by value YBottom
002010              by value XRight
002020              by value YTop
002030              by value SWP-WINDOW
002040          returning ReturnData
```

Building The Main Message Processing Routine

The second part of a PM program's Main Section involves the processing of messages received in the program's message queue. This continuous perform loop is the point at which a PM program waits for the arrival of a message and dispatches received messages to the correct window procedure. With the exception of the check for the WM-QUIT message, indicating the end of the program, there must be no other code within this message processing loop. This dispatching loop is the central point of every PM program.

Coding The WinGetMsg Call

The initial test in the loop ensures that a valid window frame was created. If no frame window was created due to an error in the Main window create call, no messages would be received and the WinGetMsg call would be held forever, effectively locking up the program.

The first call in the message-processing loop is always the WinGetMsg call. This call returns the next message in the Application Message Queue. If no message is available, the program is held in this call until a message is received in the queue. Unless message filtering is invoked, the messages will be processed on a first-in, first-out basis.

The first parameter of the WinGetMsg call is the variable that holds the program's Anchor-block handle. Parameter two is a pointer, *by reference*, to the message queue structure that will contain the message upon return from the call. Parameters three, four and five are used to filter messages from the message queue so that the application sees only selected messages.

Parameter three specifies a filter handle, the handle of a window that acts as a filter

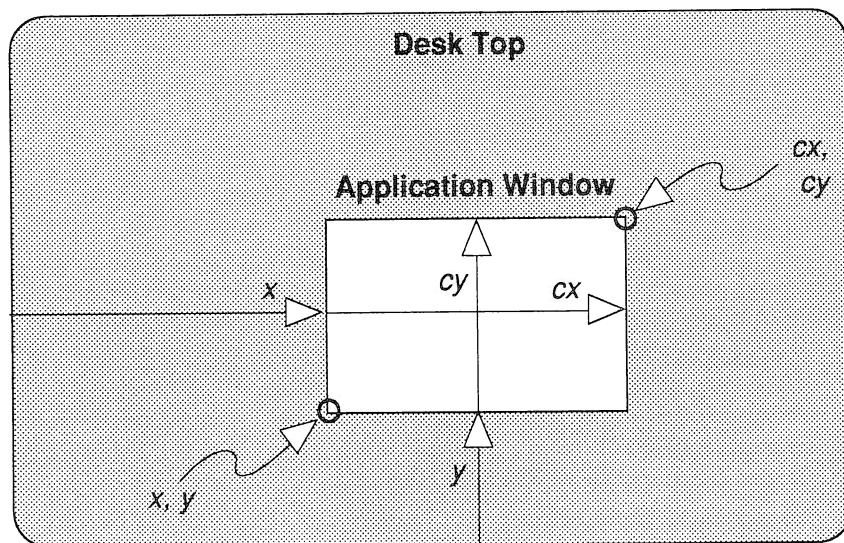


Figure 4-4 Window size and positioning points

allowing only messages for that window to be returned to the application. Parameters four and five limit messages to specific message numbers, a range from the low limit (parameter four) through the high limit (parameter five). The use of a message range limits the types of messages that are received, rather than limiting messages to a specific window as with the filter handle. Any message filtering can be dangerous, as important messages can be lost through filtering. A better practice than message filtering is to accept all messages, returning to PM those messages that are of no interest to your program. With this procedure, no key messages will be lost. Since message filtering will not be used in the sample program, these entries are coded as null.

The final parameter is the name of the variable that will receive the call's return code. Here is the WinGetMsg call used in the sample program:

```

0002090      If hwndFrame not = 0
0002100          perform until program-done
0002110
0002120          Call OS2API 'WinGetMsg' using
0002130              by value     hab
0002140              by reference QMSG
0002150              by value     LongNull
0002160              by value     ShortNull
0002170              by value     ShortNull
0002180              returning   ReturnData

```

Only the user can end a Presentation Manager program by taking an action that eventually generates the WM-QUIT message. As I discussed earlier, there are several ways that a user can signal the end of a program, but the actual WM-QUIT message can arrive in the program's message queue in only two ways.

If the user selects the *Close* entry from the System Menu or selects *Shutdown* from the Desktop Manager or presses the Alt and F4 key combination PM will post the WM-QUIT message to the program's message queue, indicating that the user wants to end the program. This version of the sample program relies upon one of these procedures to generate the WM-QUIT message.

If, however, the user selects an entry from one of the program's menus that indicates program termination, the application is notified only of the particular menu selection. It is the application's responsibility to correctly interpret the menu entry as the request to end the program and post the WM-QUIT message to its own message queue. Posting a WM-QUIT message to itself will trigger the shutdown processing exactly as if the message had come from PM.

After the WinGetMsg call completes but before the message is dispatched to a window procedure, a test must be made to determine if the message is WM-QUIT. If the WM-QUIT message was received (QMSG-MSGID = WM-QUIT), then *program-done* is set to true, the perform ends and the program enters the termination routine at the WinDestroyWindow call. If the WM-QUIT test is false, then the else routine is executed, the message is dispatched to the correct code procedure and the perform loop continues.

```
0002220      If (QMSG-MSGID = WM-QUIT)
0002230          set program-done to true
0002240      else
```

Coding The WinDispatchMsg Call

The WinDispatchMsg call is used to route received messages to the correct window procedure. The target procedure's address is not part of the dispatch call, but is determined by PM from the window handle contained in the message. PM uses the handle to determine the window's class, and from the class retrieve the window procedure address. Since WinDispatchMsg is the only way that messages may be sent to the correct window procedure, the message processing loop must end with this call. When the WinDispatchMsg call is issued, the message processing loop is interrupted and the main section held in this call until the target window procedure completes the processing of the message and returns control to this point. As a result, a return code from the window procedure is available at the conclusion of the WinDispatchMsg call.

To issue the WinDispatchMsg call, code the variable containing the handle of the Anchor-block as parameter one. Place a pointer to the message structure in parameter two. As before, specify parameter two as a pointer by including the *by reference* phrase. The variable specified in parameter three will receive the return code passed back by the window procedure at the conclusion of this call. Here is the WinDispatchMsg call and the end of the message processing loop used in the sample program:

```

002280      Call OS2API 'WinDispatchMsg' using
002290              by value      hab
002300              by reference QMSG
002310              returning    ReturnData
002320
002330          end-perform
002340      End-If.
```

Ending The Presentation Manager Program

Terminating the Presentation Manager interface is the third part of a PM program's Main Section. This routine must destroy any windows that remain open, destroy the message queue and end the program's PM interface. The PM calls used in this routine do not end the COBOL program. The program is still a COBOL program and only a STOP PROGRAM statement can end the actual program. Since a PM program can perform little productive work after terminating the PM interface, the termination of the PM interface and the ending of the COBOL program are usually found together at the end of the termination routine. But, it is important to understand that terminating the PM interface and ending the COBOL program are two distinct functions.

Only upon receipt of the WM-QUIT message can a PM program begin to shut itself down. As a result, there is usually a substantial amount of program-specific processing done at the beginning of the termination routine. The sample program contains only the calls required to perform the three PM termination functions. Additional processing, specific to each program, will need to be inserted into this skeleton termination routine. But, additional code added to this routine must not alter the order of the three termination calls. Destroying the message queue before all the windows are destroyed, for example, will result in lost messages.

The PM termination routine is the exact reverse of the initialization routine. During the initialization routine, the PM interface was created, followed by the creation of the message queue, then the creation of the first window. During the termination routine the active windows are destroyed, followed by the destruction of the message queue then the termination of the PM interface.

Coding The WinDestroyWindow Call

First, the termination routine must close all windows that remain active, even if they are in a minimized state. To destroy all remaining active windows, simply destroy the main window. PM automatically destroys all windows that are children of a parent when the parent window is destroyed. Since the Main Window is the parent of all windows created in a PM program, destroying the Main Window ensures that all other remaining windows will be destroyed.

To code the WinDestroyWindow call, place the variable containing the handle of the window to be destroyed (the Main Window) in parameter one. Parameter two must contain the variable that will receive the call's return code. Here is the WinDestroyWindow call used in the sample program:

```
002380      Call OS2API 'WinDestroyWindow' using
002390              by value hwndFrame
002400              returning ReturnData
```

Coding The WinDestroyMsgQueue Call

After destroying all active windows, the message queue must be destroyed. As with window destruction, only the handle of the message queue is passed to PM via the WinDestroyMsgQueue call to accomplish this task. Here is the WinDestroyMsgQueue call used in the sample program:

```
002440      Call OS2API 'WinDestroyMsgQueue' using
002450              by value hmq
002460              returning ReturnData
```

Coding The WinTerminate Call

The final PM call in the termination routine notifies PM that the program has completed its use of PM's services. The WinTerminate call causes PM to destroy the control blocks and free system resources allocated on behalf of the program. As with all calls in the termination routine, the handle of the item to be closed, destroyed or terminated is passed as parameter one of the corresponding call. Accordingly, to terminate the PM

interface, the Anchor-block handle is passed to PM as parameter one of the WinTerminate call. The variable that will receive the return code is coded as parameter two. Here is the WinTerminate call used in the sample program:

```
002500      Call OS2API 'WinTerminate' using
002510          by value hab
002520          returning ReturnData
```

The Window Procedure

There must be a code procedure in your program to support every window that you create. This copy of the sample program contains a skeleton window procedure containing the minimum code required for the window procedure to function correctly. All window procedures must contain this code as a minimum. Clearly, this window procedure performs no useful function, returning all messages received to PM for default processing. But, the entry statement and single PM call within this procedure are required in every window procedure.

A window procedure is no different than other COBOL subroutines that you have written. The window procedure begins with an entry statement and ends with an exit statement. The contents of a window procedure look very much like other PM code sections; they receive messages, determine which messages to process, take action upon those messages of importance and return all unprocessed messages to PM for default processing. Obviously, completed window procedures are far more complex than the one contained in this program, but this window procedure is the starting point for every window procedure you will write.

As with standard COBOL programs, every PM window procedure must begin with an entry statement. The entry statement establishes the entry point address of the called window procedure and defines the data variables passed along with the call. The variables listed in the entry statement of a window procedure must reflect the structure of a standard PM message, passed with every procedure window call. All window procedures that process standard PM messages must have the entry statement coded in this exact form. If the message time or mouse pointer coordinates are to be used by the procedure, then they must be defined after MsgParm2.

```
002640      entry 'MainWndProc' using by value hwnd
002650          by value Msg
002660          by value MsgParm1
002670          by value MsgParm2
```

In every window procedure that I write, I preset the return code value to 0 so that it needs to be changed only when an error is detected during procedure processing. This is my particular way of coding window procedures, and you may wish to implement another method. Regardless of how you choose to set the return code, it must be correctly set by the end of the procedure as the exit statement will return this value to the calling routine. Remember, the WinDispatchMsg call is held until this procedure completes, and the value specified in the EXIT statement will be returned via the WinDispatchMsg call.

The bulk of every window procedure is built around the COBOL *Evaluate* statement. The *When* phrase of this statement is used to select the messages that are to be processed, with the *When Other* phrase used to capture all unwanted messages for return to PM. The Evaluate statement requires at least one valid when phrase to be acceptable to the compiler. Since the sample program does not intercept any messages it has no when phrase. So, I have coded the individual phrases of the Evaluate statement as comments. When a valid message is to be intercepted, these comments must be removed and the statement activated.

Every message sent to your program will cause PM to take some default action if the message is returned to PM for processing. For most messages, but by no means all, PM takes no action. But, even though PM usually takes no action, you must return all unprocessed messages to PM. The *Presentation Manager Programming References* detail the exact action PM will take for each message that is returned. It is sometimes difficult to know what PM will do with a particular message, so you should review every message that your program will be returning to understand what default processing will be performed by PM.

Coding The WinDefWindowProc Call

All messages returned to PM from a window procedure must be returned using the WinDefWindowProc call. As a result, this call is always found at the end of the message evaluation procedure following the *when other* phrase.

The function of the WinDefWindowProc call is to pass unprocessed messages back to PM for default processing. To accomplish this, the WinDefWindowProc call passes the standard message data structure along with the call. As a result, to code this call simply code the standard message structure defined in the Linkage Section as the call's parameters one through four.

As with the WinDispatchMsg call, the program is held at the WinDefWindowProc call until PM completes the default processing. As a result, a return code indicating the success or failure of the default processing is available at the completion of this call. This return code is received in the variable coded as the last parameter of the call. This

variable should be the same as used by the window procedure to pass back its return codes to the calling routine. If it is, an accurate return code will be received by the calling routine both for those messages processed by the procedure and those returned to the default PM procedure.

Here is the WinDefWindowProc call used in the sample program:

```
002790      Call OS2API 'WinDefWindowProc' using
002800              by value hwnd
002810              by value Msg
002820              by value MsgParm1
002830              by value MsgParm2
002840          returning Mresult
```

Finally, the window procedure is ended with the standard COBOL exit statement, which passes a return code back to the calling routine in the Mresult variable. Here is the exit statement for the sample program's window procedure:

```
002860      exit program returning Mresult.
```

```
000010 IDENTIFICATION DIVISION.  
000020 PROGRAM-ID. COBOL-PM-SAMPLE.  
000030 DATE-COMPILED. 01-02-92.  
000040 AUTHOR. DAVID DILL.  
000050  
000060 ENVIRONMENT DIVISION.  
000070 CONFIGURATION SECTION.  
000080 SOURCE-COMPUTER. IBM-PERSONAL-SYSTEM-2.  
000090 OBJECT-COMPUTER. IBM-PERSONAL-SYSTEM-2.  
000100  
000110 SPECIAL-NAMES.  
000120      call-convention 3 is OS2API.  
000130  
000140 WORKING-STORAGE SECTION.  
000150*  
000160* Presentation Manager message structure  
000170*  
000180 01 QMSG.  
000190      05 QMSG-HWND          pic 9(9) comp-5.  
000200      05 QMSG-MSGID        pic 9(4) comp-5.  
000210      05 QMSG-PARAM1       pic 9(9) comp-5.  
000220      05 QMSG-PARAM2       pic 9(9) comp-5.  
000230      05 QMSG-TIME         pic 9(9) comp-5.  
000240      05 QMSG-POINT         pic 9(9) comp-5.  
000250      10 QMSG-X           pic 9(9) comp-5.  
000260      10 QMSG-Y           pic 9(9) comp-5.  
000270*  
000280* Window handles  
000290*  
000300 77 hab                 pic s9(9) comp-5.  
000310 77 hmq                 pic s9(9) comp-5.  
000320 77 hwndClient         pic s9(9) comp-5.  
000330 77 hwndFrame          pic s9(9) comp-5.  
000340 77 HWND-DESKTOP        pic s9(9) comp-5 value 1.  
000350 77 HWND-TOP           pic s9(9) comp-5 value 3.  
000360*  
000370* Window Classnames and titles  
000380*  
000390 77 MainWndClass        pic x(9) value "MainClas" & x"00".  
000400*  
000410* Class Style Flag  
000420*      CS-CLIPCHILDREN    0x20000000  
000430*  
000440 77 CSClass            pic 9(9) comp-5 value h"20000000".  
000450*
```

```

000460* Window Creation Flags Main Window
000470*     FCF-TITLEBAR      0x00000001
000480*     FCF-SYSMENU       0x00000002
000490*     FCF-SIZEBORDER    0x00000008
000500*     FCF-MINBUTTON    0x00000010
000510*     FCF-MAXBUTTON    0x00000020
000520*     FCF-BORDER        0x00000200
000530*     FCF-TASKLIST      0x00000800
000540*
000550 77 FCF-MAIN          pic 9(9) comp-5 value h"00000a3b".
000560*
000570* Set-Window-Position Flags
000580*     SWP-SIZE          0x00000001
000590*     SWP-MOVE          0x00000002
000600*     SWP-ZORDER         0x00000004
000610*     SWP-SHOW          0x00000008
000620*     SWP-ACTIVATE      0x00000080
000630*
000640 77 SWP-WINDOW        pic 9(4) comp-5 value h"0000008f".
000650*
000660* System Query values
000670*
000680 77 SV-CXSCREEN       pic s9(4) comp-5 value 20.
000690 77 SV-CYSCREEN       pic s9(4) comp-5 value 21.
000700*
000710* PM Message IDs
000720*
000730 77 WM-QUIT          pic 9(4) comp-5 value 42.
000740*
000750* Miscellaneous Definitions
000760*
000770 77 ReturnData        pic s9(4) comp-5.
000780     88 ReturnTrue       value 1.
000790     88 ReturnFalse      value 0.
000800 77 Loop-flag          pic x value "N".
000810     88 Program-done     value "Y".
000820 77 ShortNull         pic s9(4) comp-5 value 0.
000830 77 UShortNull        pic 9(4) comp-5 value 0.
000840 77 ULONGNull         pic 9(9) comp-5 value 0.
000850 77 NullString         pic x value x"00".
000860*
000870* COBOL Procedure pointers
000880*
000890 77 WindowProc        procedure-pointer.
000900

```



```
001360* Initialize the Presentation Manager interface
001370*
001380      Call OS2API 'WinInitialize' using
001390              by value     UShortNull
001400              returning   hab
001410*
001420* Create a message queue for this application.
001430*
001440      Call OS2API 'WinCreateMsgQueue' using
001450              by value    hab
001460              by value    ShortNull
001470              returning   hmq
001480*
001490* Register the window classes to be used.
001500*
001510      set WindowProc to ENTRY 'MainWndProc'.
001520      Call OS2API 'WinRegisterClass' using
001530          by value    hab
001540          by reference MainWndClass
001550          by value    WindowProc
001560          by value    CSClass
001570          by value    UShortNull
001580          returning   ReturnData
001590*
001600* Create the main window as a standard window.
001610*
001620      Call OS2API 'WinCreateStdWindow' using
001630          by value    HWND-DESKTOP
001640          by value    ULongNull
001650          by reference FCF-MAIN
001660          by reference MainWndClass
001670          by reference NullString
001680          by value    ULongNull
001690          by value    UShortNull
001700          by value    UShortNull
001710          by reference hwndClient
001720          returning   hwndFrame
001730*
001740* Measure the size of the dispaly screen
001750*
001760      Call OS2API 'WinQuerySysValue' using
001770          by value    HWND-DESKTOP
001780          by value    SV-CXSCREEN
001790          returning   SizeWide
001800
```

```
001810      Call OS2API 'WinQuerySysValue' using
001820              by value HWND-DESKTOP
001830              by value SV-CYSCREEN
001840              returning SizeTall
001850*
001860* Size the Main Window to 70% high by 80% wide as the desktop
001870* and center on the Desktop
001880*
001890      compute YTop      = SizeTall * .7
001900      compute XRight   = sizeWide * .8
001910      compute XLeft    = (SizeWide - XRight) / 2
001920      compute YBottom  = (SizeTall - YTop) / 2
001930*
001940* Display the Main Window
001950*
001960      Call OS2API 'WinSetWindowPos' using
001970              by value hwndFrame
001980              by value HWND-TOP
001990              by value XLeft
002000              by value YBottom
002010              by value XRight
002020              by value YTop
002030              by value SWP-WINDOW
002040              returning ReturnData
002050*
002060* Get the next message for this application or wait until a
002070* message is posted to the Application Message Queue.
002080*
002090      If hwndFrame not = 0
002100          perform until program-done
002110
002120      Call OS2API 'WinGetMsg' using
002130              by value hab
002140              by reference QMSG
002150              by value ULongNull
002160              by value UShortNull
002170              by value UShortNull
002180              returning ReturnData
002190*
002200* If WM-QUIT message received perform the closedown routine.
002210*
002220      If (QMSG-MSGID = WM-QUIT)
002230          set program-done to true
002240      else
002250*
```

```
002260* Send all messages to window default processing routine
002270*
002280      Call OS2API 'WinDispatchMsg' using
002290          by value     hab
002300          by reference QMSG
002310          returning   ReturnData
002320
002330      end-perform
002340 End-If.
002350*
002360* Destroy all remaining windows - (Main and all children)
002370*
002380      Call OS2API 'WinDestroyWindow' using
002390          by value    hwndFrame
002400          returning ReturnData
002410*
002420* Destroy the Application Message Queue.
002430*
002440      Call OS2API 'WinDestroyMsgQueue' using
002450          by value   hmq
002460          returning ReturnData
002470*
002480* Terminate this program use of Presentation Manager services.
002490*
002500      Call OS2API 'WinTerminate' using
002510          by value   hab
002520          returning ReturnData
002530 STOP RUN.
002540*****
002550
002560 MainWndProc section.
002570* * * * * * * * * * * * * * * * * * * * * * *
002580* Main Window Procedure: MainWndProc.
002590* Receives and processes all messages sent to the program's
002600* Main Window's client window. Messages are passed on
002610* entry in the standard message format.
002620*
002630* * * * * * * * * * * * * * * * * * * * * *
002640      entry 'MainWndProc' using by value hwnd
002650          by value Msg
002660          by value MsgParm1
002670          by value MsgParm2
002680*
002690* Set return code to OK and evaluate the message
002700*
```

```
002710      move 0 to Mresult
002720
002730*    evaluate Msg
002740*        when WM-XXXX
002750*
002760* Return all messages for default processing
002770*
002780*        when other
002790            Call OS2API 'WinDefWindowProc' using
002800                by value hwnd
002810                by value Msg
002820                by value MsgParm1
002830                by value MsgParm2
002840                returning Mresult
002850
002860      End-evaluate
002870      exit program returning Mresult.
002880*****
```

Chapter 5

Working With Resources

This chapter will add an Action-bar with menu items to the window created in the last chapter. Adding an Action-bar is a relatively minor task, as you will see. But, adding the Action-bar introduces the concept of a resource file and the storing of window resources outside of a program. Resource files are such an important part of PM programming, that I have allocated a separate chapter to the discussion and understanding of them. The simple task of adding an Action-bar will introduce you to every aspect of resource files, but in a very simplified and easy-to-understand format.

The Resource Script file and Resource Script Header file listings for the sample program are included with the sample program source listing in Appendix A.

Don't forget, before you begin coding the changes for this chapter, you will need to create the resource compiler command file and modify your compile and link command file by adding the *rc* command to bind the Resource file to your application. See Chapter 3 for a complete description of both of these command files. For a more complete discussion on using the Resource compiler, see the *Presentation Manager Programming References*, part of the Developer's Toolkit.

A Word About Resources

Certain items used in the construction and manipulation of windows are referred to as resources. Resources are not windows, but collections of read-only character-based tables and strings or bit map-based images used in the construction of windows. Window titles, prompts and warning messages, menus and menu items, accelerator key tables and dialog box templates are all examples of character-based resources. Window icons, bit map images, and screen pointers are all examples of bit map resources. Normally,

you would code these items within your program's Working-Storage section, where they would become part of the data segment of your compiled program. But as resources, these items are stored outside of your program's data segment. They reside in their own data segment and are either bound with the program's .EXE file or reside separately in a dynamic link library, loaded and accessed by the program during execution.

Resource files are restricted to read-only data items. Consequently, these items may not be altered during execution allowing OS/2 greater flexibility in managing the memory they occupy. As read-only data, OS/2 simply discards resources when memory becomes constrained then reloads them when they are needed by a program. There is no need to save read-only resources in the Swapper file, and that significantly cuts the amount of time OS/2 requires to manage memory. If the resources are contained within a dynamic link library, they can be shared by multiple programs and the initial loading of dynamic link library resources can be deferred until they are required to save additional memory. A COBOL's data segment can't be managed in this way as it normally contains updated variables.

A Resource file is initially created as an ASCII file, in a manner similar to the creation of a COBOL source file. This ASCII resource source file is known as the Resource Script file and has the file extension of *rc*. Prior to its use by a program, the Resource Script file must be compiled into an object file, called a Resource file, with the file extension of *res*. After compiling, the resource file must bind into either a dynamic link library or with your executable program (see Figures 3-1 and 3-2).

There are four distinct advantages to using resource files.

- 1) Keeping resources out of your program's data segment makes your programs smaller and allows more room for non-resource data items.
- 2) As read-only items, resources may be shared among many applications, reducing data redundancy and the time and complexity required to maintain these items.
- 3) As read-only segments, OS/2 can manage these resources with greater efficiency than normal data segments.
- 4) By separating these types of data from the program, they may be modified without changing, recompiling or relinking the programs that use them.

You are not required to use resource files to create PM programs. Everything that you place in a resource file can be placed into your COBOL program's Working-Storage section; the resulting programs will function in exactly the same manner. However, the advantages of using resource files are so great that you should use them with every PM program that you write. The sample program will use Resource Script files extensively.

For simplicity's sake and to let you concentrate on building and using resource files, the initial versions of the sample program's resource files are bound into the sample program's executable file. While this limits the system benefits of using resource files, it does not change how they are built or used within a PM program. Building and managing resources in a dynamic link library is discussed in Chapter 16.

Coding Conventions Unique to The Resource Script File

The Resource compiler is a C Language-based program, which means the Resource Script file must be written using C Language notation. But, to make the process easier, many of the more esoteric C Language notations have been replaced with keywords. For example, BEGIN and END replace the paired { } symbols. But two major coding conventions must be followed.

First, comments must be placed within paired /* */ identifiers. Comments are free form, may go anywhere within the file and may start anywhere on a line. As an example, the comment "THIS IS A COMMENT" should be coded as follows:

```
/* THIS IS A COMMENT */
```

See the Resource Script files within Appendix A for more examples of the use of comment lines.

Second, the C Language allows the use of the underscore character (_) within labels, and the Presentation Manager programming convention has adopted the use of the underscore character rather than the dash character (-) in labels. So, to be consistent, use underscores in the Resource Script files and Resource Header files, but change them to dashes in the COBOL declarations.

Creating The Resource Script File

There are seven different types of text, text strings and bit map data that may be included within a resource file. They are:

- Bit maps
- Dialog and Window templates
- Fonts, Icons and screen pointers
- Keyboard accelerator tables
- Menu tables
- String tables
- User-defined resources

Rather than describe each resource type in detail in this chapter, I will discuss the individual resource items as they are implemented in the sample program. This will allow each chapter to concentrate on a different resource, making it easier to see how each type is used. In this chapter, I will discuss the Menu resource used to create the Action-bar.

The first entry in every Resource Script file must be a reference to the C Language header file, OS2.h. This file supplies the C Language definitions that are required by the Resource compiler. Additional header files are optional but are usually included to define the numeric values of the individual resources. In accordance with C Language notation, the Resource compiler is instructed to include header files through the use of the #include statement rather than the copy statement used with the COBOL compiler. Header files carry a .h file extension.

The use of separate header files for user-defined data is not mandatory. All Resource Script file definitions may be combined into one header file, or header files can be ignored entirely and the user definitions placed directly into the Resource Script file. For the sample program, I have built a separate header file for each Resource Script file. To identify which header file supports which Resource Script file, I have given each header file the same file name as its corresponding Resource Script file. But, regardless of how you structure your compiler definitions, remember that the OS2.h header file is required to support the Resource compiler. Here are the two header files used in this chapter's SAMPLE.RC Resource Script file.

```
#include    "os2.h"
#include    "sample.h"
```

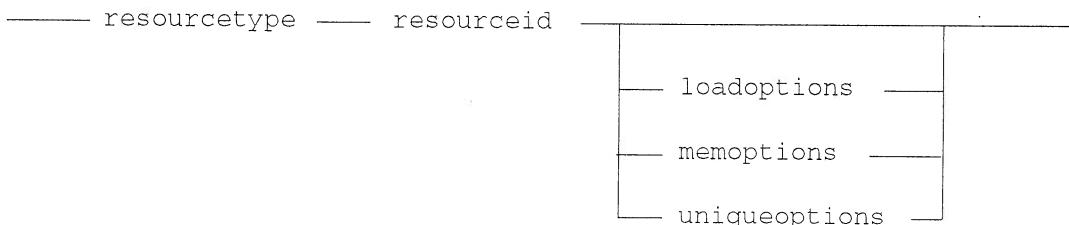
After the OS2.h and any optional header files are listed, begin defining the resources to be included in your script file. You must start each resource definition by specifying the resource type. If you are building a multiline resource, or resource table, then BEGIN and END statements must follow the resource type statement and surround the individual entries.

There are eleven different resource types to support the seven general resources. You may include as many of them as necessary to completely define your resources and multiple occurrences of the same resource type may be specified. Each of the resource types is fully explained in the *Presentation Manager Programming References*. The following are the eleven resource types that can be used within a Resource Script file.

RESOURCE TYPE DEFINITIONS

ACCELTABLE	A table of accelerator keys used to convert keystroke combinations into menu commands.
BITMAP	A custom bit map to be displayed on the screen.
DLGINCLUDE	An include file containing the dialog templates to be incorporated into the resource file at compile time.
DIALOGTEMPLATE/ WINDOWTEMPLATE	A table of dialog or window templates.
FONT	A file containing a font.
ICON	An include file holding a bit map image to be used as a program icon.
MENU	A table defining the contents of an Action-bar and/or menu.
POINTER	An include file holding a bit map to be used as a screen pointer.
RCDATA	The definition of simple number or character string.
RESOURCE	An application defined resource.
STRINGTABLE	A table defining one or more complex strings.

Here is the format to use when coding the resource type statement:



The resourceid is the numeric value that becomes the identifier of the resource. Each resource is known to PM by this numeric value and when referencing a resource, your program must use this resource identity value.

Loadoptions defines to OS/2 when the resource should be loaded. PRELOAD indicates that the module should be loaded immediately, in other words, at the start of program execution. LOADONCALL indicates that the resource should only be loaded when it is called by the application. The load options are mutually exclusive.

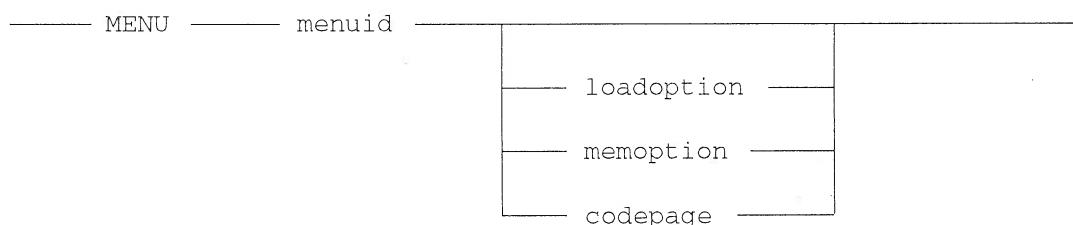
Memoptions define how much flexibility OS/2 has in managing the resources once they have been loaded into memory. FIXED indicates that, once loaded, the resources must remain at a fixed memory location. The MOVEABLE memory option indicates that the resource module may be moved by OS/2 if memory is being compacted. The DISCARDABLE memory option indicates that the resources may be discarded when no longer needed.

Uniqueoptions vary with each resource type statement. You should review the *Presentation Manager Programming References* for each resource type that you use to insure that any unique options are correctly specified.

Defining The Menu Entry

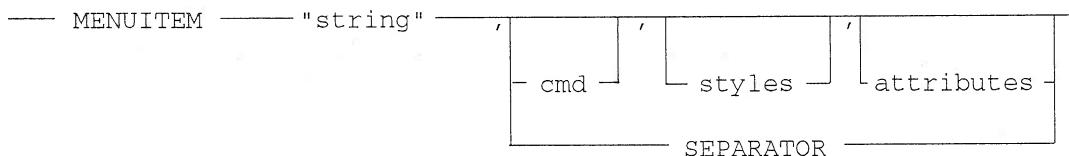
To create a menu entry, you must first define the resource as a menu, using the MENU statement. As with most resource type definition statements, this statement defines the resource type, the assigned numeric resource identifier, the load and memory management options and an optional codepage to be used for this resource.

For the sample program the PRELOAD load option is specified as the menu must be available when the Main Window is first created. There is no benefit to delaying the loading of this resource. No memory options were specified because the default MOVEABLE is correct for this resource. For a more complete description of the MENU and MENUITEM options, see the *Presentation Manager Program References*. Menu statements have the following standard format.



After entering the menu statement, the individual menu entries are listed using the MENUITEM statement in the order they are to appear on the Action-bar. Individual menus must be bracketed with BEGIN and END statements so the Resource Compiler

can determine the limits of each menu. Here is the format to use when coding MENUITEM statements:



The first entry defines the character string that will appear on the Action-bar and must be enclosed in quotes. A tilde character (~) may be placed before any character within the string to designate the following character as a mnemonic character. The mnemonic character is displayed on the menu as an underlined character and when the Action-bar, or menu, is active, pressing the mnemonic character invokes the menu entry exactly as if the mouse had been used for the selection.

The cmd entry specifies the numeric value that will be returned to the window procedure, via the WM-COMMAND, WM-SYSCOMMAND or WM-HELP message, when this item is selected by either the mouse or the mnemonic character key.

The styles entry contains one or more style options ORed together with the | operator to define the overall style of this menu entry. The C Language | operator may be used when creating resources because the Resource compiler is a C Language compiler. Here are the MIS- values that may be used to define a menu item's style.

MENU STYLE VALUES

MIS-SUBMENU	This item is a submenu, that is, selection of this item will cause another menu to be displayed.
MIS-SEPARATOR	This menu item is displayed as a horizontal line dividing the menu into multiple parts.
MIS-BITMAP	This menu item is a bit map.
MIS-TEXT	This menu item is a text string.
MIS-BUTTONSEPARATOR	This menu item is a button.
MIS-BREAK	This menu item is the last item displayed in a row or column. The next item starts a new row or column.
MIS-BREAKSEPARATOR	This menu item draws a separator line and ends a row or column.
MIS-SYSCOMMAND	When this menu item is selected, the procedure is notified via the WM-SYSCOMMAND message rather than the usual WM-COMMAND message.
MIS-OWNERDRAW	This menu item is drawn by the owner and not by PM.
MIS-HELP	When this menu item is selected, the procedure is

	notified via the WM-HELP command rather than the usual WM-COMMAND message.
MIS-STATIC	This menu item is for display only and cannot be selected by the user.

Like the menu styles, the menu attributes entry contains one or more attribute options ORed together with the | operator to define the overall attributes of this menu entry. Here are the MIA- values that may be used to define a menu item's style.

MENU ATTRIBUTES

MIA-HILITED	The menu entry will be displayed in a reverse color when it is selected by the user or the procedure.
MIA-CHECKED	A check mark will appear next to the menu item when it is selected by the user or the procedure.
MIA-DISABLED	This menu item cannot be selected by the user and is displayed in a gray color, indicating its non-selectable state.
MIA-FRAMED	A frame is drawn around this menu item.
MIA-NODISMISESS	When this menu item is selected, the menu containing this entry will not be hidden before the application window is notified.

Here are the MENUITEM statements used in the sample program's Resource Script file to define the Action-bar entries.

```

MENU      ID_MainWind
BEGIN
  MENUITEM  "~File",     ID_File
  MENUITEM  "~Edit",     ID_Edit
  MENUITEM  "~View",     ID_View
  MENUITEM  "~Options",   ID_Option
  MENUITEM  "~Window",   ID_Window
  MENUITEM  "~Help",     ID_Help
END

```

Creating The Resource Header File

Once you create the Resource Script file, you must assign numeric values to each of the labels included within the file. This is normally done in a C Language header file referenced at the start of the Resource Script file. Placing the numeric values in a separate header file keeps the Resource Script file cleaner and makes it easier to read. In the sample program, I used the header file sample.h. Remember, you are dealing with

C Language notation, so header files must have a file extension of H.

The individual values assigned to each label are entirely up to you. The values are stored as short integers so you may select values up to 65,535. However, to make your program more readable, there should be some reasoning to the values selected. For the sample program, I selected numbers from 100 to 199 for the all menus, with each Action-bar entry increasing by ten. As the Action-bar entries have submenus added to them in later chapters, the submenu entries will use the nine numbers following the related Action-bar entry. Here is the Resource Script header file for the sample program:

```
#define ID_MainWind      10
#define ID_File          110
#define ID_Edit          120
#define ID_View          130
#define ID_Option         140
#define ID_Window         150
#define ID_Help           160
```

Compiling The Resource Script File

The completed Resource Script file must be compiled before it can be bound to, and used by the sample program. Complete information on compiling the Resource Script file and binding it with the application is contained in Chapter 3. If you have built the BUILD.CMD command file as described in Chapter 3, executing this file is sufficient to compile the Resource Script file and bind it to the existing COBOL executable file. Thereafter, the Resource compiler stage 2 command in the MAKE.CMD file will bind the latest version of the resource file to the program every time it is recompiled.

Modifying The COBOL Program

Because the Action-bar entries are contained within a resource file, adding these entries to the sample program requires only three simple changes to the sample program.

First, the Frame Create flags must be modified to specify the inclusion of an Action-bar window within the sample program's frame window. Since the value of the Frame Create flags variable is the binary sum of the individual flags, simply add the value of the FCF-MENU flag (0x00000004) to the *FCF-MAIN* variable. Adding this value increases the value of the FCF-MAIN variable to x["a3f"](#). So, to include the Action-bar, simply set the

FCF-MAIN variable to the hex value h"00000a3f". Here is the updated FCF-MAIN declaration in the sample program:

```
000910 77 FCF-MAIN      pic 9(9) comp-5 value h"00000a3f".
```

Second, you must define all referenced resource values within the Working-Storage Section of your program. Note that you do not need to include all of the values defined in the resource file, only those that you will be referencing. This is especially important with shared resource files that often contain hundreds of items your program will never use. For example, a shared resource file of messages may contain hundreds of messages, of which your program will use only a few. Defining only the resources that you will use saves a significant amount of space in your program's data segment.

Although it is not required, good programming convention dictates that the same labels be used in both the resource file and the COBOL program. Remember, when declaring the resource labels within your COBOL program, the underscore character (_) must be replaced with the dash character (-). In the sample program, the resource label *ID_MainWind* becomes the COBOL label *ID-MainWind*. These resources are declared within the COBOL program's Working-Storage as unsigned short integers, defined as pic s9(4) comp-5. Here is the one resource file definition used in this chapter's sample program:

```
001510 77 ID-MAINWIND    pic 9(4) comp-5 value 10.
```

Third, a change is required to the WinCreateStdWindow call to notify PM that you are using a Resource file and to identify the individual resource to be used with this window. Since this resource file is bound with the sample program, there is no separate dynamic link library module to identify to PM. Accordingly, parameter seven, the resource module's identifier must be coded as a null. Parameter eight is the identity of the individual resources that are to be associated with this window. Enter the label of the variable holding the resource identifier as parameter eight. This variable's label should be, but does not need to be, the same as the resource label of the menuitem entry. For the sample program, all resources identified with the value of ten (*ID-MainWind*) will be associated with this window. Here is the modified WinCreateStdWindow call with the resource identifier specified:

```
005590    Call OS2API 'WinCreateStdWindow' using
005600                by value      HWND-DESKTOP
005610                by value      ULONGNull
005620                by reference FCF-MAIN
005630                by reference MainWndClass
005640                by reference NullString
005650                by value      ULONGNull
005660                by value      USHORTNull
005670                by value      ID-MainWind
005680                by reference hwndClient
005690                returning    hwndFrame
```

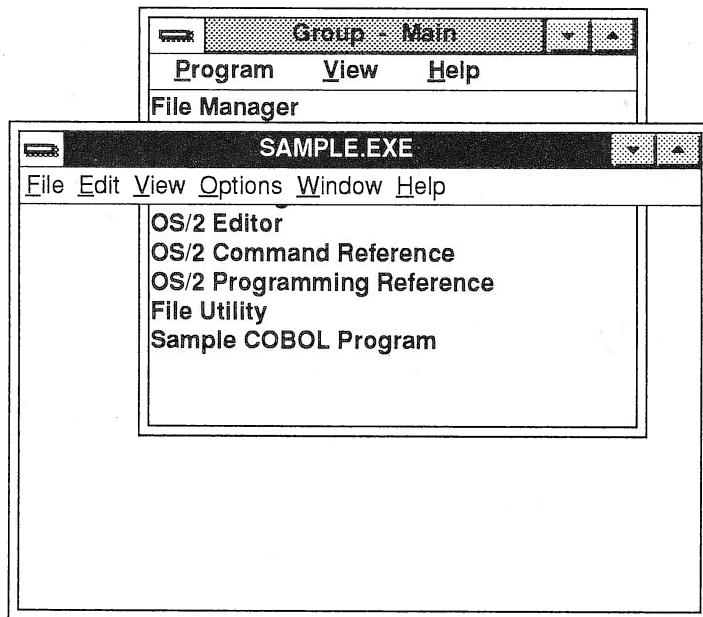


Figure 5-1 The Sample Program's Main Window with the Action Bar (Version 1.3)

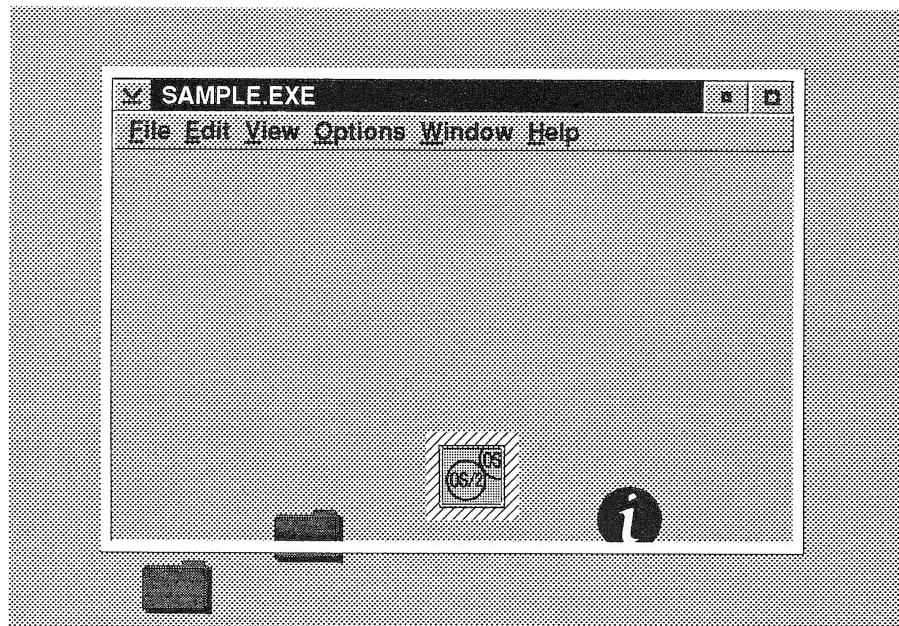


Figure 5-2 The Sample Program's Main Window with the Action Bar (Version 2.0)

Chapter 6

Processing PM Messages

If you will pardon the pun, with the Presentation Manager, the message is the medium. Messages are the primary means of interprocess communications within the Presentation Manager session. Nothing happens within the Presentation Manager session that does not cause at least a few messages to flow. And, in many cases, more than just a few. Depending upon how the user is interacting with the system, a window procedure can be inundated with hundreds of messages. The sheer volume of messages makes an efficient message processing structure mandatory.

A Word About Messages

Messages can originate from three different sources. User interaction with the keyboard and mouse will generate messages, as will most calls to OS/2 or PM, and a process can generate messages to another process and to itself.

A keyboard message is generated for every keyboard action taken by the user and is routed to the window procedure supporting the focus window. For each key pressed or released, a WM-CHAR message is generated. The WM-CHAR message contains the keyboard control codes, the repeat count, the hardware scancode, the ASCII character translation and the virtual key translation, if a virtual key was defined. Here are the keyboard control codes used to transmit special key status within the WM-CHAR message. See Appendix D for the hexadecimal value of each of these codes.

KEYBOARD MESSAGE CODES

KC-CHAR	The ASCII translation in MsgParm2 is valid.
KC-SCANCODE	The hardware scancode in MsgParm1 is valid.

KC-VIRTUALKEY	The virtual key translation in MsgParm2 is valid.
KC-KEYUP	The key moved in an upward direction. If this flag is off, the key moved in a downward direction.
KC-PREVDOWN	This key was down for previous key strikes. If this flag is off, the key was not down for previous key strikes.
KC-DEADKEY	This key, by itself, has no meaning for this window.
KC-COMPOSITE	This key must be combined with the previous dead key to determine the character code.
KC-INVALIDCOMP	This key is not a valid combination with the prior dead key.
KC-LONEKEY	This key was depressed and released without any other key action between the down and up movement.
KC-SHIFT	The shift state was active when this key was pressed.
KC-ALT	The ALT state was active when this key was pressed.
KC-CTRL	The CTRL state was active when this key was pressed.

If an accelerator key table containing an entry for the pressed key is defined for the focus window, then an additional WM-COMMAND, WM-SYSCOMMAND or WM-HELP message will be generated containing the command value defined for this key in the accelerator key table.

A Mouse message is generated whenever the mouse pointer is moved or the user presses or releases one of the mouse buttons. Unlike keyboard messages that are sent only to the focus window, a mouse movement message is sent to each window the mouse pointer moves across, regardless of the focus. A single movement of the mouse pointer from one screen location to another may result in mouse movement messages being sent to several window procedures. Every time the mouse pointer moves from one set of coordinates to another, a WM-MOUSEMOVE message is generated. The WM-MOUSEMOVE message contains the current x and y coordinates of the screen pointer in relation to the lower left corner of the window.

The mouse button messages, WM-BUTTONxUP, WM-BUTTONxDOWN and WM-BUTTONxDBLCLK, indicate which of the three mouse buttons was moved, the direction of its movement, and if the button movement was a single-click or double-click.

OS/2 and PM messages are generated to supply information the user has requested or to indicate changes in the OS/2 or PM environment. Dynamic Data Exchange, clipboard, focus change, paint, set focus, semaphore and timer messages are all examples of these operating system generated messages. While unique in the origin, these messages are processed in exactly the same manner as user-generated or application-generated messages.

Messages generated by a process are most often used to interact with window controls. Process messages are used to pass commands affecting the appearance or operation of window controls, extract data from window controls or query the status of window

controls. But process messages are also used to pass information between threads within the same process or between procedures within the same thread.

There are two distinct ways in which messages may be passed. Messages may be sent directly from one procedure to another or posted by one procedure to the message queue of another procedure. Sent messages are similar to a COBOL call in that the sending procedure is suspended and remains suspended until the procedure receiving the message returns control. Posting a message, however, suspends the posting procedure only until the message is placed into the receiving procedure's message queue, allowing both the posting and receiving procedures to continue executing, independently and simultaneously. See Figure 9-1.

Figure 6-1 shows the logic used to process a message. It is important to remember that every time a call is made to OS/2 or PM, the calling procedure is suspended until the called procedure returns control. Here is the logic used to process messages within a Presentation Manager program.

- 1) A message is removed from the program's message queue using the WinGetMsg call.
- 2) The message is passed to the proper window procedure using the WinDispatchMsg call.
- 3) The window procedure may choose to intercept and process the message. Message processing usually involves multiple PM API calls.
- 4) The window procedure may choose to ignore the message and return it to PM for default processing using the WinDefWindowProc call.
- 5) After completing one of the two logic paths, the window procedure returns to the main message processing loop at the WinDispatchMsg call.
- 6) The message processing loop returns to the WinGetMsg call to process the next message or wait for the next message to be received.

Since many of the Presentation Manager calls require the handle of the target window, I should point out that at any time there may be more than one active window supported by a window procedure. I mentioned in an earlier chapter that each window procedure supports a single class of windows, but that a class may have a single window, multiple windows, multiple iterations of a single window or multiple iterations of multiple windows. Multiple windows will present a huge problem if the window procedure tries to keep track of each window and associate incoming messages and data with the correct window. As it turns out, PM contains the solution for this potential problem. Each

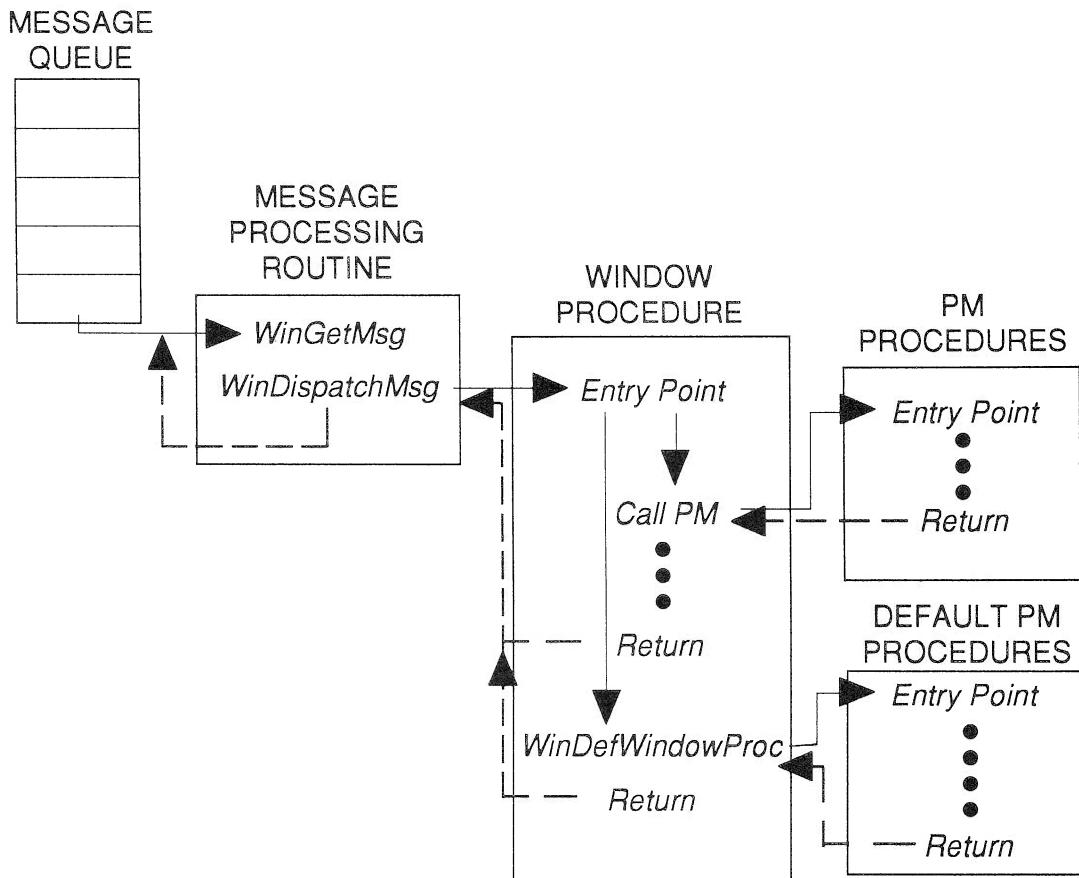


Figure 6-1 PM application message processing flow

message contains the handle of the target window. By simply coding the Linkage Section definition of the message handle field in all PM calls that require the handle of the target window, the correct window is always specified. Traditionally, you will see most PM calls within a window procedure coded with *hwnd* for the handle of the target window.

Message Recursion

Figure 6-1 shows the processing logic for a single message. But, a single message is not the norm for PM programs; multiple messages at various stages of processing are the most likely situation for a PM procedure. The most common cause of multiple messages within a program is message recursion, the process of one message generating additional

messages to the same procedure.

The processing of messages usually results in one or more PM calls being issued. A significant number of these calls will generate messages that are routed back to the same procedure that issued the PM call. For example, when a window procedure wishes to obtain the focus, it issues the WinSetFocus call. This call, in turn, generates a WM-SETFOCUS message that will be routed back to the procedure that issued the WinSetFocus call. Since the first message is suspended at the WinSetFocus call waiting the completion of the call, the procedure begins to process the WM-SETFOCUS message. This message will be processed until a PM call suspends the processing of this message. At this point, the procedure may begin to process a third message. Theoretically, there is no limit to the number of messages that a procedure may be processing at any point in time.

The need to support message recursion requires that all window procedures be written as reentrant procedures. A reentrant procedure is written in such a way that the processing of one message will not impact the processing of another message. High on the list of reentrant code no-nos is the use of global variables and switches. The use of global variables and common switches has a profound and distinctly negative affect upon window procedures. Reentrant window procedures should restrict the use of variables to those defined in Local-Storage, and input should be restricted to the data passed in the message parameters.

How To Check For Selected Messages

Each PM message has a defined numeric value and this value is carried within the message as the message ID. The message ID is used as the match criteria to select messages for processing. For simple messages, checking the message ID is all that is required to correctly select messages for processing. But, most PM messages are not simple and require additional examination beyond the message ID before the processing decision can be made.

To make handling and talking about messages easier, each message value has been equated to a name and it is this name by which a message is usually referred. So, for example, PM programmers refer to the WM-PAINT message, rather than message number 35, or the WM-SETBORDERSIZE message rather than message 68. The message name implies the message's function in a way the message number never can.

Using the when phrase of the COBOL Evaluate statement, the message ID of each message received is examined by the window procedure and those messages with IDs to be processed are automatically selected. It is not necessary to understand or process every message received by the window procedure. If the message is not one of the specific

messages the procedure is looking for, then it is of no interest to the procedure. Many of the messages received do not directly effect your program and PM often routes undocumented messages to itself via the program's message queue. All messages not specifically selected for processing are thus captured by the when other phrase and returned to PM for default processing using the WinDefWindowProc call.

For example, to process all WM-PAINT messages the when WM-PAINT phrase is included within the window procedure's evaluation process. Then, whenever a WM-PAINT message (message 35) is received, the code within the when WM-PAINT phrase is executed.

If you are writing the sample program by chapters, the introduction of the evaluation statement will require a small change to your existing code. The COBOL evaluate statement, carried in the Chapter 4 sample program as comments must be activated and the WinDefWindowProc call, moved within the evaluation process and placed at the end of the window procedure after the when other phrase. Failure to include both the when other phrase and the WinDefWindowProc call at the end of every window procedure will result in unprocessed messages and unpredictable results.

Processing Messages For A Window

The WinDispatchMsg call in the Main Section sends messages to the window procedure specified for the class of the window whose handle is in the first message field. The WinDispatchMsg call passes the message via the Linkage Section and transfers control to the procedure at its specified entry address. For example, in the sample program any message with the handle of the Main window's client window (*hwndClient*) is dispatched to the *MainWndProc* window procedure. This procedure supports all windows of the class of *MainWndClass*, which includes the Main window's client window.

The first statement in any window procedure must be the COBOL Entry statement specifying the entry point of the procedure and identifying the Linkage Section variables passed to the procedure. Remember, variables are passed to procedures via the Linkage Section so only the Linkage Section message structure may be specified within a window procedure, not the QMSG message structure defined in the Working-Storage Section

At the beginning of every procedure, I always include an optional statement to preset the procedure's return code to zero. This assures that unless it is changed, a zero return code will be passed back to the calling routine upon completion of this message processing. This is the way I code my procedures, you may choose to code your procedures differently.

The next statement of every window procedure must be the COBOL Evaluate statement, followed by as many when phrases as required to intercept all messages of interest to the

procedure, one when phrase for each different message to be intercepted. The last phrase of the Evaluate statement must be when other with the WinDefWindowProc call as the only function for this phrase. You may choose to add additional code to your procedures, but the following message evaluation structure must appear in every window procedure that you write.

```

evaluate Msg

    when WM-MSGA
        •
        •
        •
    when WM-MSGB
        •
        •
        •
    when WM-MSGC
        •
        •
        •
    when other
        Call OS2API 'WinDefWindowProc' using
    End-evaluate.
exit program returning Mresult.

```

This chapter will introduce the code required to intercept and process messages. Obviously, the exact actions taken when a message is processed varies with each message and each program, but the structure used to intercept messages is standard for every COBOL PM program. Writing this relatively simple message processing routine will hopefully give you a clearer picture of exactly how messages are intercepted and processed or returned to PM.

This chapter will add the processing of the WM-PAINT message. The WM-PAINT message is generated by PM whenever all or part of a window has been corrupted and needs to be repainted on the desktop. Moving, resizing, overlaying, uncovering, drawing into and creating a window are all functions that trigger the WM-PAINT message. Processing the WM-PAINT message will finally fill the hole in your Main window's Client window.

To accomplish the painting of the sample program's Main window's client window, the sample program must intercept and process every WM-PAINT message sent to the MainWndProc. The application must update and repaint its windows every time they are altered by external forces. The default window processing procedure for WM-PAINT will not update the window's contents. The lack of WM-PAINT support in the default procedure is evidenced by the fact that Chapters 4 and 5 returned WM-PAINT messages to PM, but no action was taken to paint the Main window's Client window.

The process of painting a window introduces three new PM calls that perform this basic function. In order to paint a window, however, several functions must be performed, including the allocation of a presentation space, before the actual painting can occur.

A Word About Presentation Spaces

Window painting is the first of the Graphics Programming Interface (Gpi) functions to be used in the sample program. All graphic APIs use a special area for graphics drawing called a presentation space (PS). The presentation space can be thought of as a universal drawing pad that accepts all graphical output from a program, eliminating the need for a program to understand the specifics of each graphical output device it might be using. The device independence provided by the presentation space means that a program sees all graphical output devices, displays, printers, plotters, etc., in exactly the same way, and that means a program requires only a single drawing routine to support all graphical devices. This device independence allows one image to be sent to different devices with different densities and characteristics without the need to alter the image for each of the devices.

After the graphical images are drawn into the presentation space, a data structure called a Device Context is used to translate the presentation space images into a form compatible with the output device. The device context holds all of the device-specific information for the output device, isolating it from the presentation space and the program. The correct device context must be linked to the presentation space by the program before the images can be moved from the presentation space to the actual output device. The device context data structure is used by the output device's presentation driver to translate the device-independent drawing data in the presentation space into device-dependent output data as the images are moved to the actual output device.

There are two types of presentation spaces available to a program, regular presentation spaces and micro presentation spaces. The major differences between a regular PS and a micro PS are the size of the drawing space, the complexity of the image to be held within it and the amount of time that the space is retained.

For very complex images, where images are to be sent to multiple output devices, or when images are to be held for relatively long periods of time, perhaps while it is manipulated by the user, a normal presentation space must be obtained from PM using the GpiCreatePS call. Normal presentation spaces must be destroyed when the output is completed, using the GpiDestroyPS call.

When the images are relatively simple or when the images are not to be retained once they are drawn, a micro presentation space should be used. Like normal presentation spaces, micro presentation spaces must be created using the GpiCreatePS call and

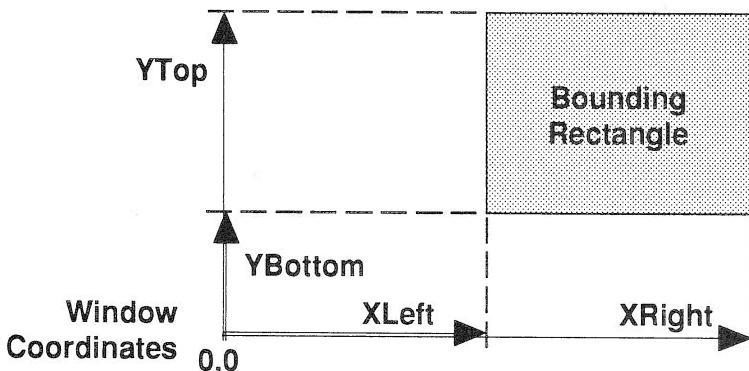
destroyed using the GpiDestroyPS call.

For window output only to a display device, the Window Manager maintains a predefined group, or cache, of micro presentation spaces paired with a video device context. These predefined cached micro presentation spaces are allocated quickly when temporary window output is required. There are only a small number of cached presentation spaces and they are a shared system resource, so they should never be held for long time periods.

The WinBeginPaint function is an easy way to begin graphical programming because the underlying cached micro presentation space and video device context are automatically allocated by the WinBeginPaint call and released by the WinEndPaint call. So, while it may seem that a presentation space and device context are not required to paint a window, be assured that they are there.

Coding The WinBeginPaint Call

Parameter one of the WinBeginPaint call requires that handle of the target window. This handle allows PM to associate the device context for the cached micro presentation space with the correct window. The handle of the target window is the same as the message handle. Simply code the message handle variable from the Linkage Section message structure as parameter one. Parameter two must be either the handle of a regular presentation space obtained with a prior GpiCreatePS call or a null to tell Presentation Manager to allocate a cached micro presentation space. Since a cached micro presentation space is correct for a simple painting of a window, parameter two is coded as a null. Parameter three is a pointer to a data structure that defines the area to be painted, in window coordinates. Called the bounding rectangle, this area defines the coordinates of the four sides of the rectangle. These coordinates are measured from the lower left corner of the window, coordinates 0,0. The bounding rectangle allows the program maximum flexibility in determining exactly which part of a window to paint. Where windows contain complex images, repainting only that portion of the window that is invalid saves a significant number of CPU cycles. Since the entire Main window's client window needs to be repainted, it is not necessary to determine the exact rectangle that needs to be repainted. To cause the entire window to be repainted, the bounding rectangle coordinates must be set to 0. Since the bounding box coordinates are Local-Storage variables, their contents cannot be assumed to be 0 and must be set prior to the call. The last parameter is the handle of the cached micro presentation space returned at the completion of this call.



Notice that the variable for the returned presentation space handle (*hps*) is not defined with the other handles in Working-Storage, but in Local-Storage. It is possible through recursion that the window procedure could be concurrently processing messages that used presentation spaces for more than one window. If the presentation space handle variable were in Working-Storage, the multiple routines might overlay each other's handles causing an action to be taken against the wrong window. By defining this variable in Local-Storage a separate variable will be allocated for each iteration. But, be aware that this handle will not be retained across multiple messages. Thus, the presentation space must be created, used, then destroyed in a single message cycle.

Here is the WinBeginPaint call used in the sample program:

```

009550      Move 0 to XLeft YBottom XRight YTop
009560      Call OS2API 'WinBeginPaint' using
009570          by value      hwnd
009580          by value      LongNull
009590          by reference rcl
009600          returning     hps

```

Coding The WinFillRect Call

With the bounding rectangle (*rcl*) set to 0 to indicate that the entire Main window's client window should be painted, the WinFillRect call fills the rectangle with any one of thousands of colors. For the sample program, I chose white (defined as a value of -2). The color is set by moving the value -2 to the local variable *color*. The declaration *WEMWindowColor* will eventually become part of a dialog data passing area. So, if you are coding the sample program by chapters, temporarily code this variable as a signed, short integer with the value -2. (pic s9(4) comp-5 value -2).

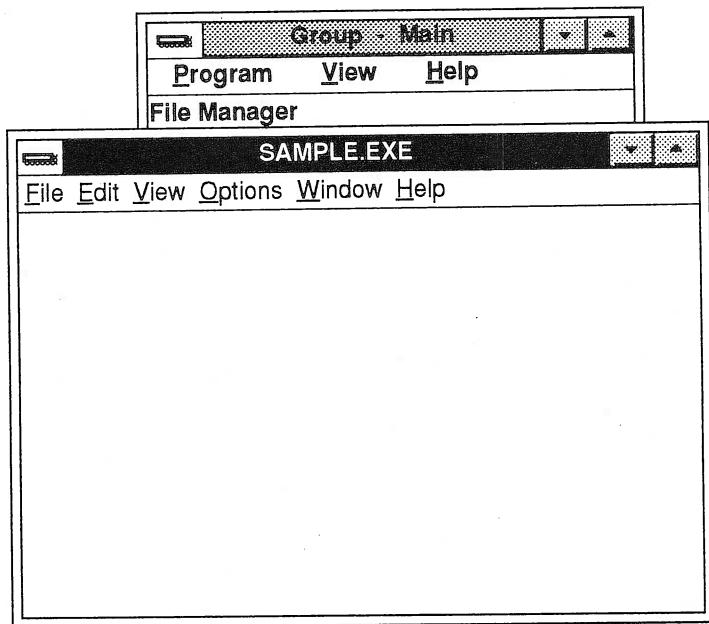


Figure 6-2 The Sample Program's Main Window with a painted Client Window (Version 1.3)

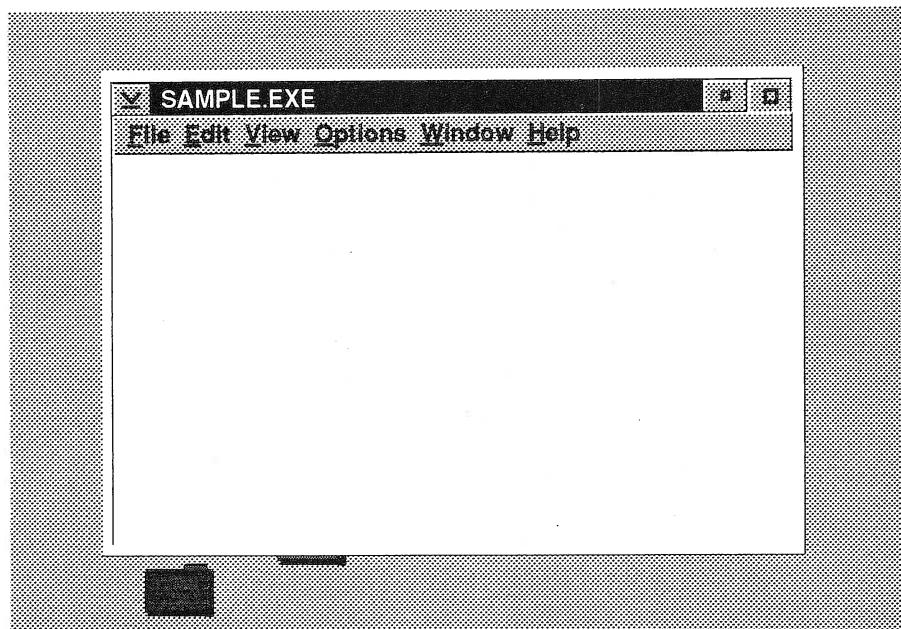


Figure 6-3 The Sample Program's Main Window with a painted Client Window (Version 2.0)

The handle used in WinFillRect call is not the Main window's client window handle, but the handle of the cached micro presentation space. Remember, you draw into the presentation space and PM makes the conversion using the display device context and updates the screen.

To code the WinFillRect call, place the name of the variable holding the handle of the cached micro presentation space in parameter one. Place a pointer (*by reference*) to the Local-Storage bounding rectangle structure in parameter two and the name of the variable holding the desired color in parameter three. The last parameter is the name of the variable that will receive the call's return code. Here is the WinFillRect call used in the sample program:

```
009620      Move WEMWindowColor to Color
009630      Call OS2API 'WinFillRect' using
009640          by value      hps
009650          by reference rcl
009660          by value      Color
009670          returning    ReturnData
```

Coding The WinEndPaint call

The final call in the window paint routine, WinEndPaint, notifies PM that the window painting is complete, and that the cached micro presentation space may be released back to the Window Manager. To code the WinEndPaint call place the label of the variable containing the handle of the cached micro presentation space in parameter one. Parameter two is the label of the variable that is to receive the call's return code. Here is the WinEndPaint call used in the sample program:

```
009690      Call OS2API 'WinEndPaint' using
009700          by value      hps
009710          Returning    RetrunData
```

Chapter 7

Beginning The User Dialog Process

This chapter begins a rather lengthy discussion of the user dialog process. Dialogs are, quite simply, conversations between the program and the user. Dialogs are at the heart of most PM programs. But, don't confuse the formal Presentation Manager dialog process, as described in this and following chapters, with a generic two-way conversation, often referred to as a dialog. Standard Presentation Manager windows can be made to have the appearance and function of a dialog, but they are not dialogs. Selecting an entry from a menu might be considered a two-way conversation, but that does not turn menus into dialogs. The distinction is important because many of the rules for dialogs apply only to the formal dialog procedures, not the simple display of data in a window.

Dialogs may be as simple as a one-line message or as complex as multiple interrelated windows, but they always have these common properties:

- Dialogs are always initiated by the user but started by the program.
- The end of a dialog is always initiated by the user but the dialog procedure ends the dialog.
- Dialogs always involve a two-way conversation between the user and the dialog procedure.
- Data entered into a dialog by the user may be changed or discarded anytime prior to the end of the dialog.
- The user determines the order of dialog control selection. The dialog procedure must respond to the user's pattern of selection.

Dialogs are composed of two parts, a frame window, called a dialog window, containing one or more dialog control windows used to interact with the user, displaying information to the user and collecting information from the user, and the a code procedure that supports the conversation. While this may sound like the structure of a standard window, standard windows do not support the complex two-way conversations that dialogs do, and do not require the more sophisticated code procedures that support dialogs.

The Message Box

The message box is a very specialized type of dialog. Because it is so specialized, it is also the simplest type of dialog to understand and program. The message box is a good place to begin learning about dialogs.

The message box performs the single function of displaying a message and returning the ID of the button selected by the user. In return for limiting the functions performed by the message box, PM assumes all of the work usually performed by the dialog procedure. By limiting the actual message box to a simple window with only a message and a few predefined push buttons, the user is not required to define a dialog template before invoking a message box. As a result of PM assuming so many of the traditional dialog functions, a message box is created, displayed, maintained and ended, using the single WinMessageBox call. No additional code is required within the program.

The content of a message box is limited to the message box frame, a single message, an optional icon and one to four push buttons. The message to be displayed in a message box must be passed to PM as part of the WinMessageBox call. The number and function of the push buttons along with the icon to be displayed are determined by the message box style flags ORed into a single style word and passed to PM as part of the WinMessageBox call. Here is a list of the message box style flags that controls the number and meaning of the message box buttons. See Appendix D for a complete list of message box style flags.

MESSAGE BOX STYLES

MB-ABORTRETRYIGNORE	Abort, Retry, Ignore
MB-CANCEL	Cancel
MB-ENTER	Enter
MB-ENTERCANCEL	Enter, Cancel
MB-HELP	Help
MB-OK	OK
MB-OKCANCEL	OK, Cancel

MB-RETRYCANCEL	Retry, Cancel
MB-YESNO	Yes, No
MB-YESNOCANCEL	Yes, No, Cancel

Here is a list of the icon style flags that determines which icon is displayed on the message box:

ICON STYLE VALUES

MB-CUANOTIFICATION	No Icon
MB-NOICON	
MB-ICONQUESTION	
MB-QUERY	
MB-CUSWARNING	
MB-ICONEXCLAMATION	
MB-WARNING	
MB-ICONASTERISK	
MB-INFORMATION	
MB-CRITICAL	
MB-CUACRITICAL	
MB-ERROR	
MB-ICONHAND	

Here are the style flags that determine the default button and the modal style of the message box.

MESSAGE BUTTON STYLES

MB-DEFBUTTON1	Button 1 is the default button
MB-DEFBUTTON2	Button 2 is the default button
MB-DEFBUTTON3	Button 3 is the default button

MESSAGE MODALITY VALUES

MB-APPLMODAL	The message box is application model
MB-MOVEABLE	The message box is moveable via the system menu
MB-SYSTEMMODAL	The message box is system model

You will note the sample program does not process the button ID returned by the WinMessageBox call. The About message is an informational message with no

meaningful data returned by the user, so no processing is required. The single button in the About message box is present only to allow the user to indicate when they are finished with the message box. If, however, processing were required, the returned button ID must be queried to determine which button the user had selected and what action should be taken by the program. Here are the return codes from the WinMessageBox call that indicate which button the user selected.

MESSAGE BOX RETURN CODES

OK	MBID-OK	1
CANCEL	MBID-CANCEL	2
ABORT	MBID-ABORT	3
RETRY	MBID-RETRY	4
IGNORE	MBID-IGNORE	5
YES	MBID-YES	6
NO	MBID-NO	7
HELP	MBID-HELP	8
ENTER	MBID-ENTER	9
function not successful	MBID-ERROR	65,535

Adding Pull-Down Menus

Dialogs are always initiated by the user and started by the program. So, to allow the user to initiate the About dialog a pull-down menu containing the About menu entry must be added to the sample program's Action-bar.

Pull-down menus, like the Action-bar, are resources and adding them to a window is accomplished entirely through changes to the resource file. Of course, subroutines must be added to the program to process any commands generated by these pull-down menu items.

Two resource statements are used to define the actual contents and structure of a menu. The SUBMENU statement defines an Action-bar or menu entry that has a lower-level menu beneath it. A SUBMENU entry may have additional SUBMENU entries below it, but for every SUBMENU entry there must ultimately be at least one MENUITEM entry. The MENUITEM statement defines the lowest level of a menu, that menu that presents the actual list of items from which the user must choose. The format of the MENUITEM statement is described in Chapter 5. Here is the format of the SUBMENU statement:

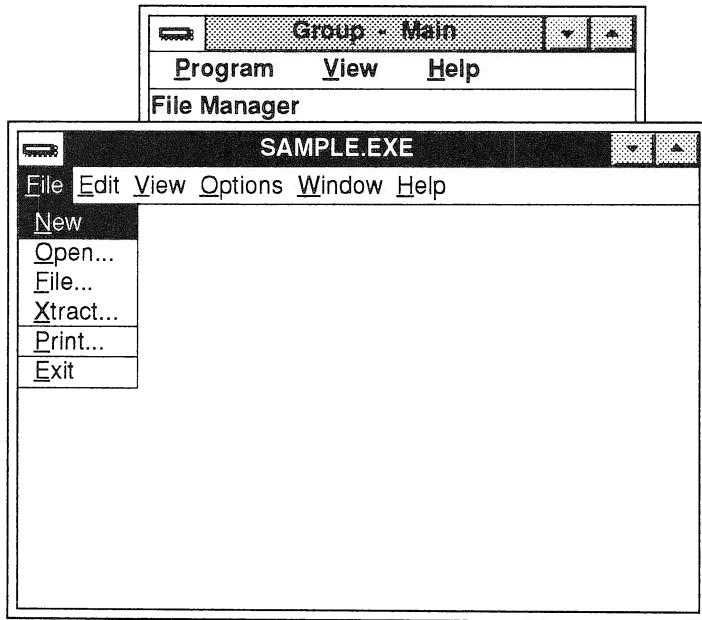
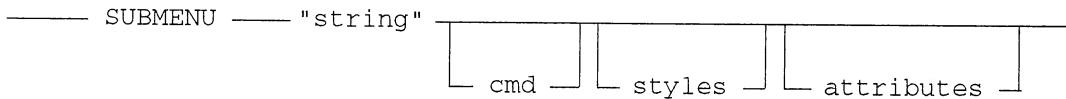


Figure 7-1 The Sample Program's Main Window with the File pull-down menu (Version 1.3)



The first parameter defines the text string that will appear on the menu. This entry must be enclosed within quotation marks. As with a MENUITEM statement, a tilde (~) character may be placed prior to any character in the string to indicate that character as the mnemonic character.

The **cmd** parameter specifies the numeric value that will be returned to the window procedure, via a WM-COMMAND, WM-SYSCOMMAND or WM-HELP message, when this entry is selected by the user.

The **styles** parameter contains one or more style options that are ORed together with the | operator to define the style of this SUBMENU entry. See Chapter 5 for a list of the valid style entries.

The **attributes** entry contains one or more attribute options that are ORed together with the | operator to define the attributes of this SUBMENU entry. See Chapter 5 for

a list of menu attributes.

Each successive lower-level group of SUBMENU or MENUITEM statements must be set off with paired BEGIN and END statements. Here is the structure that must be used to set multiple level menus within a Menu entry:

```
Menu      ID_Menu
BEGIN
    SUBMENU "String 1", ID_Submenu1
    BEGIN
        MENUITEM "String 2", ID_Menu2
        MENUITEM "String 3", ID_Menu3
        MENUITEM "String 4", ID_Menu4
    END
    SUBMENU "String 5", ID_Submenu5
    BEGIN
        MENUITEM "String 6", ID_Menu6
        MENUITEM "String 7", ID_Menu7
    END
END
```

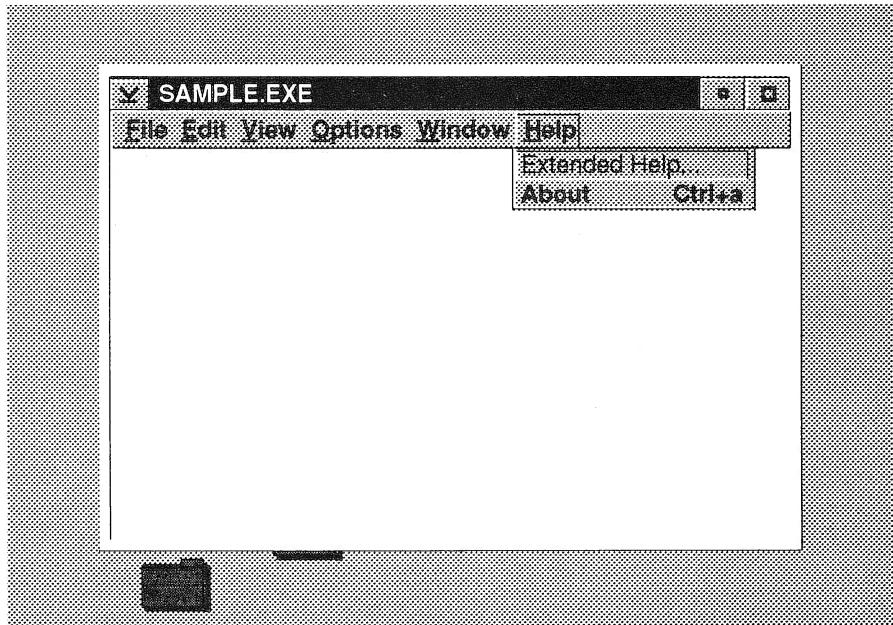
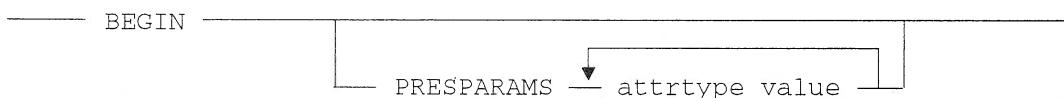


Figure 7-2 The Sample Program's Main Window with the Help pull-down menu (Version 2.0)

Any BEGIN statement may contain the optional presentation parameters phrase that is used to change the colors, font face or font sizes of entries within this BEGIN and END paired group. The PRESPARAM phrase is composed of an attribute type paired with a value for the attribute. There may be as many sets of attributes and values as necessary. Here is the structure of the BEGIN statement. See Chapter 12 for more information about presentation parameters. See Appendix D for a complete list of presentation parameters.



Modifying The Resource Script File

When the second level pull-down menus are added to the sample program, the Action-bar entries added in Chapter 5 no longer represent the lowest-level menu. So, the current MENUITEM statements in the Resource Script file must be converted to SUBMENU statements and the new pull-down entries inserted as MENUITEM statements. This causes the Resource compiler to treat the new entries as pull-down menus rather than additional items for the Action-bar.

Following each SUBMENU statement, MENUITEM statements are inserted to form the actual pull-down menu. Remember to bracket each set of MENUITEM statements with BEGIN and END statements. Additionally, add two commas indicating the lack of style flags followed by the MIA_DISABLED attribute to each MENUITEM statement, except the About entry. The MIA_DISABLED phrase causes the menu item to be disabled so that it cannot be selected by the user. Disabled items are displayed using light gray characters, indicating their non-selectable status to the user. As the routines to process the commands generated by these menu item are added in later chapters, the MIA_DISABLED keyword are removed, making the item available for selection by the user.

If a third level menu will be generated by the current menu item then PM automatically adds a right-pointing arrow to the menu item. If the current menu entry will cause a dialog to appear requesting additional input from the user before the menu item is initiated, three periods must be added to the end of the menu text string indicating to the user that further input is required before the requested menu item can be acted upon.

Here are the revised menu entries for FILE SUBMENU on the Action-bar. The file menu is shown in Figures 7-1 and 7-2. See the Resource Script file listing in Appendix A for the completed Main Window Action-bar and menus.

```

SUBMENU "~File", ID_File
BEGIN
    MENUITEM "~New",           MI_New,,   MIA_DISABLED
    MENUITEM "~Open...",      MI_Open,,   MIA_DISABLED
    MENUITEM "~File...",      MI_File,,   MIA_DISABLED
    MENUITEM "~Xtract...",    MI_Xtract,, MIA_DISABLED
    MENUITEM SEPARATOR
    MENUITEM "~Print...",    MI_Print,,  MIA_DISABLED
    MENUITEM SEPARATOR
    MENUITEM "~Exit...",     MI_Exit,,   MIA_DISABLED
END

```

Each MI_ label used in a MENUITEM statement represents the numeric value that will be passed to the program via a WM-COMMAND, WM-SYSCOMMAND or WM-HELP message when the item is selected. These numeric values must be defined to the Resource compiler within the Resource Script file itself or, more commonly, within a header file. For the sample program, I chose values that related to the values of the Action-bar SUBMENU statements, but any unused value may be specified. See the Resource Header file in Appendix A for the #define statements used in the sample program.

Finally, for each value defined in the Resource Script file that is to be processed within the program, a similar declaration must be added to the Working-Storage Section of the program. As you did in Chapter 5, use the same labels in the Working-Storage Section, but be sure to change the underscore character (_) to a dash character (-).

Accelerator Keys

The user should always be given the option of using a keystroke or a combination of keystrokes to activate a menu item, rather than requiring the use of the mouse. This function is supported in PM through the use of mnemonic characters and accelerator keys.

A mnemonic key is a single character within a menu entry - designated by a preceding tilde (~) within the resource string and displayed with an underscore character - that activates the menu entry. The mnemonic key is an alternative to menu selection by the mouse. Mnemonic keys are active only when the menu containing the entry is displayed. This is a good news, bad news situation. By activating the mnemonic keys only when the menu is displayed, these keys may be used as input to the window when the menu is hidden. The bad news is that experienced users must display the menu holding the

desired entry before they have a keyboard alternative to the mouse. For many experienced users, having to wade through menus can be an annoyance.

Accelerator keys, or key combinations, are in many ways the reverse of mnemonic keys. Accelerator keys are active whenever a window has the focus, not just when the proper menu is displayed. Accelerator keys avoid the need to open one or more menus before pressing the mnemonic key. As long as the window has the focus, all the accelerator keys for that window are active.

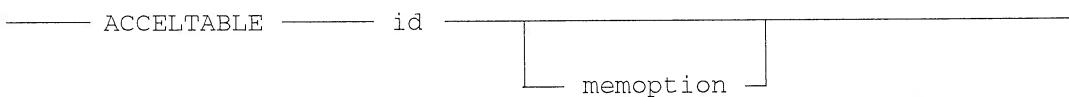
As with mnemonic keys, accelerator keys present a good news, bad news scenario. For some users, avoiding menus altogether may improve their performance. However, once an accelerator key is defined, it may not be used for any other purpose with its related window. Because of this, care must be taken in assigning accelerator keys or key combinations to avoid duplications across menus and to ensure that the selected keys have no meaning to the application.

When accelerator keys are supported for a menu item, a text representation of the key or key combination must be added to the menu item to identify the key or key combination to the user. To add the accelerator key or key combination to the menu item, insert a "\t" at the end of the menu item text string, followed by the text indicating the accelerator key or key combination. The \t symbol causes the menu entry to be divided into two columns with the accelerator key text left-aligned in the right-hand column. For the sample program, "\tCtrl+a" is added to the end of the About menu item text string to indicate that the key combination to activate the About Message Box is the Control key together with the lower-case letter A. Figure 7-2 shows how the Help pull-down menu looks with the addition of the About menu accelerator key text. Here is the modified About menu item in the sample program's Resource Script file:

```
MENUITEM "<About\tCtrl+a", MI_About
```

Adding The Accelerator Key Table

The accelerator key table is another of the ASCII string tables that is stored in the Resource file. The ACCELTABLE statement identifies the resource and assigns the resource identifier coded as parameter one. The use of resource identifiers allows different accelerator key tables to be defined for different windows. The accelerator key table added to the sample program's resource file in this chapter will be used with the Main Window so the identifier *ID_MainWind* is used as the resource ID. Here is the format to use when coding the ACCELTABLE statement:



The memory option defines how much flexibility OS/2 has in managing the resource once it is brought into memory. FIXED indicates that the resource must remain at a fixed memory location. The MOVEABLE memory option indicates that the resource module may be moved by OS/2 if memory is being compacted. The DISCARDABLE memory option indicates that the resource may be discarded when it is no longer needed.

As with the other resource tables, individual table entries must be bracketed between paired BEGIN and END statements. The individual table entries begin with the accelerator character code. This is either a constant or a quoted character. If a quoted character is used, the CHAR acceloption is assumed. The second parameter is the command value returned in the WM-COMMAND, WM-SYSCOMMAND or WM-HELP message when the Main Window has the focus and the user presses this key combination. The final parameter contains one or more accelerator options, separated by commas, that modify or further define the key specified in parameter one. You may specify as many accelerator options as necessary to correctly define the key combination, with the exception of the restriction listed below.

Here is the format to be used when coding the individual ACCELTABLE entries and the list of accelerator key options.

```
_____ keyvalue _____ cmd _____ acceloption _____
```

ACCELERATOR KEY OPTIONS

ALT	The Alt key must be down for the accelerator key to be active.
CHAR	Specifies a character is to be generated.
CONTROL	The Ctrl key must be down for the accelerator key to be active.
HELP	The key will be passed in the WM-HELP message.
LONEKEY	The accelerator key alone will generate the WM-COMMAND message.
SCANCODE	The scancode of the accelerator key will be returned, not the key character.
SHIFT	The Shift key must be down for the accelerator key to be active.
SYSCOMMAND	The key will be passed in the WM-SYSCOMMAND message.
VIRTUALKEY	A virtual key will be returned in place of the key character.

The CHAR, LONEKEY, SCANCODE and VIRTUALKEY options are mutually exclusive.

In the sample program, the lower-case letter A is used to activate the About message box. But use of the letter A alone would prevent the lower-case letter A from being used elsewhere within the Main window. So to isolate this accelerator key and allow the lower-case letter A to be used elsewhere within the window, the CONTROL key is required along with the lower-case letter A key. Only when the CONTROL key is depressed at the time the lower-case letter A is pressed, will the About message box be displayed. To specify this key combination, the accelerator key entry has the lower-case letter A within quotation marks, together with the option CONTROL. Enclosing the letter within quotation marks causes the Resource compiler to treat A as a letter rather than an ASCII value. The command message *MI_About*, coded as parameter two, will be generated when the accelerator key combination is pressed. Using the same command value specified in the menu table allows a single message processing routine to support the About message box regardless of how the user initiates the request.

Here is the initial accelerator table used in the sample program's Resource Script file:

```
ACCELTABLE ID_MainWind
BEGIN
    "a", MI_About, CONTROL
END
```

For an accelerator key table to be active, the related window must have the frame create flag FCF-ACCELTABLE specified.

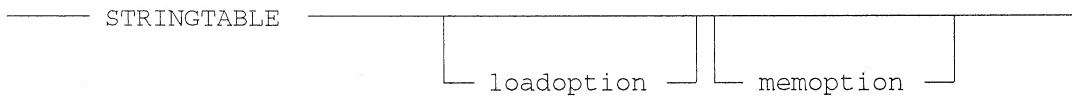
Adding The Resource File String Table

Together with menus and accelerator tables, the Resource file is the repository of the text strings. By storing a program's text strings outside of its data segment and referencing them via a resource identifier, there is maximum flexibility in modifying a program's text without performing traditional program maintenance. Changing the content of a text string is as easy as replacing the old string with the new string in the Resource Script file, recompiling the resource and binding the new resource to the program.

The string table is a multiline resource table. As with other multiline tables, the STRINGTABLE statement is followed by the actual strings, bracketed between paired BEGIN and END statements. The STRINGTABLE statement identifies the resource, but unlike the other resource tables, the STRINGTABLE statement has no associated resource identifier. The individual strings within the table are considered the resource, not the table itself, so the individual string entries have the resource identifiers, not the table. By making each string a resource, strings can be shared by all windows within

a single thread, or if the strings are within a dynamic link library, all threads within a process or all processes within the PM session.

Here is the format to be used when coding the STRINGTABLE statement. The load and memory options are the same as those used with prior resource tables.



The actual string entries consist of a resource identifier, integer value or a label defined as an integer, followed by the ASCII text string within quotation marks. The string text itself may contain any ASCII character and there are provisions for entering special control characters through the use of an escape symbol. Here are the special control characters that may be included within a text string:

\a	Bell character
\t	Horizontal tab character
\nnn	An ASCII character in Octal notation
\dnn	An ASCII character in Hex notation

Here is the initial string table used in the sample program's Resource Script file:

```
STRINGTABLE
BEGIN
    IDS_MsgBoxOne, "This is the COBOL PM sample program"
END
```

Accessing String Data In The Resource File

All resource objects, including text strings, are stored outside the program's data segment and beyond the addressability of the program. So before your program can reference these resource objects, they must be moved from their read-only storage into the program's Working-Storage. The WinLoadString call is used to load character strings into a program's Working-Storage. The sample program shows how this is done with the message box text string.

Before the WinLoadString call can be issued, however, several variables must be defined. A buffer is needed to hold the returned text string and an unsigned short integer is needed to hold the size of the returned string. An additional unsigned short integer is needed to pass to PM the maximum size of the string that may be returned.

```

002140 77 MsgBoxLength      pic s9(4) comp-5 value 0.
002150 01 MsgBoxText.
002160    05 MBoxText       pic x(80).
002170    05 filler         pic x value x"00".
003390 01 MaxStringLength  pic s9(4) comp-5.

```

Coding The WinLoadString Call

To code the WinLoadString call, place the variable holding the Anchor-block handle in parameter one. Parameter two is used to identify the dynamic link module that contains the resource, if one is used. As the sample program's resources are bound with the program this parameter must be set to null. Parameter three is the label of the variable holding the identifier of the resource string to be returned. This is the value assigned to the string in the resource string table. Parameter four is the label of the variable containing the maximum length string that may be returned. This should be the size of the buffer that will receive the string and should be large enough to hold any message returned plus the null-termination character. Parameter five is a pointer *by reference* to the buffer that is to receive the returned resource string. The length of the resource string is returned as the call's return code and is available in the variable specified in the *returning* phrase. Here is the WinLoadString call used in the sample program:

```

013160          Move 80 to MaxStringLength.
013170          Call OS2API 'WinLoadString' using
013180            by value    hab
013190            by value    ShortNull
013200            by value    IDS-MsgBoxOne,
013210            by value    MaxStringLength,
013220            by reference MsgBoxText
013230            returning   MsgBoxLength

```

The returned resource string will have a null termination character appended. Since this string will be returned to PM in the WinMessageBox call, the null termination character must remain as part of the message text.

Coding The WinMessageBox Call

Parameter one of the WinMessageBox call is the handle of the parent of the message box. The parent handle determines the location and placement of the message box on the screen. The message box will be centered within the parent window and clipped to fit the window if it is larger than the parent window's current size. To ensure that the message box is always visible and large enough to allow the message to be read completely, specify the desktop as the parent of the message box and code *HWND-DESKTOP* as parameter one. Parameter two is the handle of the owner of the message box.

As with other owner-windows, this is the handle of the window procedure that will receive messages related to the message box. However, for message boxes, PM calculates the actual owner at the time the message box is created. PM determines the actual owner by searching up the parent hierarchy, starting at the window passed in parameter two of the WinMessageBox call, until a child of the window specified as the parent in parameter one is found. If such a window exists, it is made the actual owner of the message box. If such a window does not exist, then the owner of the message box is set to NULL. The Main window's Client window, as part of the Main window, is a child of the desktop, so the variable holding the Main window's Client window handle is coded as parameter two.

Parameter three is a pointer to the buffer holding the text string to be used in the message box. Parameter four is a pointer to a text string to be used as a message box title. For the About message box, no title is used so parameter four is coded as a null string. Parameter five is the message box window identifier. This is an integer value that identifies the message box to the help system. This number is passed to the HK-HELP hook if a WM-HELP message is generated for the message box. The use of unique values allows the help messages to be customized to each message box. For the sample program, I used the value of one represented by the working storage declaration of *ID-One*. The sixth parameter defines the style of the message box and the number and labels of the buttons appearing at the bottom of the box. This information must be defined prior to the WinMessageBox call. As with other style flags, these values are defined by ORing together the desired individual message box style flags. The last parameter, the call's return code, is the ID of the button selected by the user.

Here is the WinMessageBox call used to display the About message in the sample program:

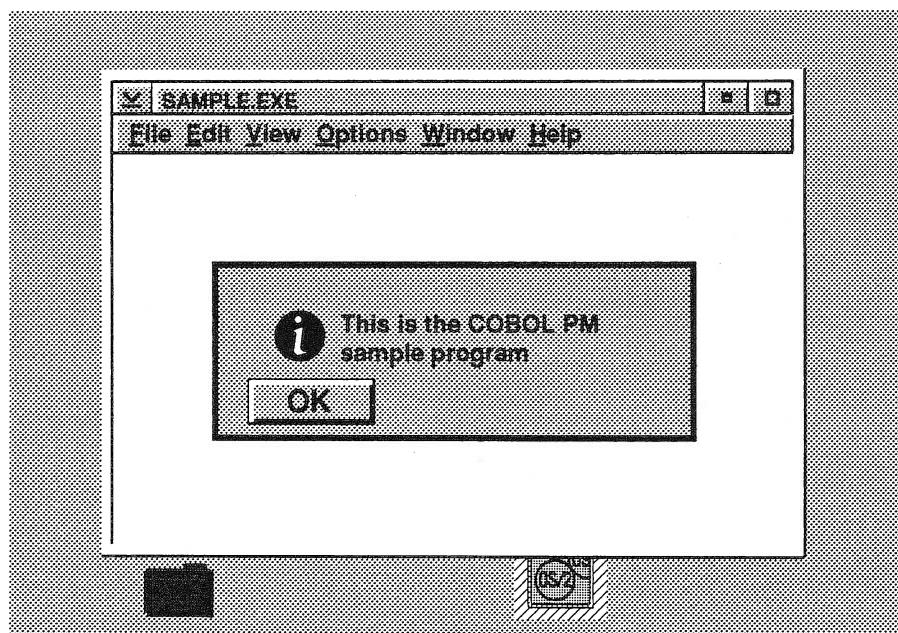


Figure 7-3 The About message box (Version 2.0)

```

013250      Call OS2API 'WinMessageBox' using
013260          by value      HWND-DESKTOP
013270          by value      hwndClient
013280          by reference MsgBoxText
013290          by reference NullString
013300          by value      ID-One
013310          by value      MB-INFO-OK
013320      returning     Mresult

```

Processing The WM-COMMAND Message

Chapter 6 introduced message processing with the relatively simple WM-PAINT message. With messages such as WM-PAINT, the message ID is sufficient to select the message for processing. The WM-COMMAND message, however, is more complex and more typical of PM messages. The WM-COMMAND message can have as many different meanings as there are command values within the program.

With complex messages, such as WM-COMMAND, the two message parameters must

be examined to understand the exact meaning of the message. Each message parameter is four bytes long and the data structure of each parameter depends upon the individual message. Message parameters can be complex. The WM-COMMAND message has a single unsigned short integer containing the command value in the left-most two bytes of parameter one, an unsigned short integer containing the source of the command in the left-most two bytes of parameter two and a Boolean value, indicating whether the mouse or keyboard initiated the command, in the right-most byte of parameter two. Because the location of the data and the data type vary by message, you must be familiar with each message you process to ensure that the correct parameters and parameter locations are checked. Here is the format of the WM-COMMAND message that carries the MI-About command:

WM-COMMAND MESSAGE				
Message ID	MsgParm1w1	MsgParm1w2	MsgParm2w1	MsgParm2w2
(Hex) 2000	A200	0000	0300	0000
(Dec) 0032	0162	0000	0003	0000
WM-COMMAND ID				
MI-About command (ushort)				
Unused (ushort)				
Command posted by an accelerator key				
Message is posted as a result of a keystroke (False)				

As with all data within an Intel-based computer, message data is stored in a reversed format. In the above message, the hexadecimal data is shown in the exact format received from PM. The decimal data is shown as reversed by COBOL (based upon the comp-5 phrase). You must remember when checking message parameters that COBOL will always reverse the data so you must be careful to check only the bytes within a parameter that hold the data. If for example, you were to examine the left two bytes of the long integer MsgParm1 for the message command, you would find them empty because COBOL reverses the long integer and generates the value 0000000162. If, however, you correctly examine the left short integer of message parameter 1, MsgParm1w1, you would find the correct message command 00162.

Coding The Command Routine

Within the window procedure, the COBOL Evaluate statement is used to select messages using the message identifier. When a selected message contains more than one command, individual parameters must be examined until the exact message is determined. To locate the MI-About message, the short integer MsgParm1w1 of each

WM-COMMAND message must be checked for the *MI-About* command. The exact method of examining message parameters is determined by the contents of the message parameters. Here is how the *MI-About* command message is processed in the sample program:

```
009750      when WM-COMMAND  
  
013150          If MsgParam1w1 = MI-About  
                  •  
                  •  
                  •  
013330          End-if  
  
013370      when other
```


Chapter 8

Icons And Mouse Pointers

Until now, if you minimized the sample program's Main window, all you see is a small white square at the bottom of the display screen. Lacking a window icon, the Presentation Manager displays the lower left-hand corner of the window's client window whenever that window is minimized. Displaying this small portion of the window does a poor job of identifying the minimized window or program to the user. So, to better identify the sample program, or any program, when it is minimized, a graphical symbol representing the program is added to the program. This graphical symbol is called an icon.

An icon is a bit map image that represents a choice, status, window or program to the user. Within PM there are many uses for icons.

The most common use of an icon is the representation of a minimized window or inactive program. Representing a window or a program as an icon allows a user to move quickly among windows, regardless of their size or location on the desktop. Icons offer the user infinitely more flexibility in manipulating large numbers of windows by allowing the user to focus on those windows of current interest and minimize, or remove, those windows that are of no current interest.

Icons are frame window resources and as such a program can have a different icon for each frame window. For a more dramatic effect, window icons may be changed while they are being displayed so that they can continually reflect the current status of the window or program they represent.

Another common use of icons is the communication of special status to the user. The type and importance of error messages or the current status of a dialog, window or the system can often be communicated more quickly and easily via an image than a message. There are several predefined icons supplied as part of the PM system that may be combined with custom icon images to represent whatever special status requirements your program may have.

The mouse pointer is another of the PM bit map images. The pointer's primary function is to display the current location of the mouse on the desktop. But like other PM features, the pointer has taken on additional functions. Pointers have become an effective way to display the current status or function of a window. The Version 2.0 *clock face* symbol, for example, not only shows the current location of the mouse but indicates that the window temporarily cannot accept input. Pointers have been used to represent the currently active drawing tool, the contents or function of a window, a class of control windows, a data entry error or just about anything that can be better represented by a graphical image than a message. By manipulating images in quick succession, movement can be added to pointers for even greater clarity and understanding. You will certainly want to utilize custom pointers in your programs.

Creating An Icon Or Mouse Pointer

Icons and mouse pointers have an identical structure so they are both created using the Icon Editor, part of the OS/2 Developer's Toolkit. During installation of the Toolkit, the toolkit editors are added to your system as the *Toolkit Editors* group for Version 1.3, or as a *Toolkit* object for Version 2.0. To start the Icon Editor, select the *Toolkit Group* or *Toolkit* object from the Desktop Manager, then select the *Icon Editor*. While it is not my intention to give you detailed instructions on the use of the Icon Editor, a few words of advice may be helpful. Unless stated otherwise, the following suggestions apply to both icons and mouse pointers.

When you start the Icon Editor, the display type is set to the display type currently in use. If you edit an existing icon or pointer object, the display type is automatically changed to match the density of the object being edited. But, if you are creating a new mouse or pointer object, you should pay close attention to the display type specified for the object.

Drawing an object on a display with a density different from the display used to run the object can cause the drawn object to be adjusted by PM. To adjust the pel density, PM arbitrarily eliminates selected rows and columns, every other row and column when reducing a 64 x 64 pel object to a 32 x 32 pel object. It is not unusual to see one-pel-wide rows and columns disappear when this adjustment is made, depending upon their location within the object. Sometimes the most carefully drawn icon is adjusted into an almost unrecognized form. The following chart shows the pel densities and number of colors available for each display adapter when using the Version 2.0 Icon Editor.

Display Adapter	Image Pel Density	Number of Colors
XGA	64 x 64	256
8514	40 x 40	256
8514 (16 colors)	40 x 40	16
8514 (1.2 format)	64 x 64	16
VGA	32 x 32	16
VGA (1.2 format)	32 x 32	16
EGA	32 x 32	16
CGA	32 x 16	2
Independent 1.2 format	64 x 64	16
Independent 1.1 format	32 x 32	2

Always set the device type to the display adapter that will be used at execution time. If you have a mixed display environment, then you have two options. You can draw a separate image at the correct pel density for each display adapter that will be used. Then when the program is executed, query the system using the WinQuerySysValue call, with the SV-CXICON, SV-CYICON or SV-CXPOINTER, SV-CYPOINTER parameters, to determine the pel density of the current display adapter and load the corresponding icon or mouse pointer. Your second option is to draw all of your icons and mouse pointers at the pel density supported by most of your displays, or pick a density that stands up to the conversion process the best. The most compatible density for all displays seems to be 32 x 32 pels. But, you should experiment with the best density for the images that you create. Here are some additional hints about drawing icons and mouse pointers that may be useful to you in making the decision about how to create icons and mouse pointers.

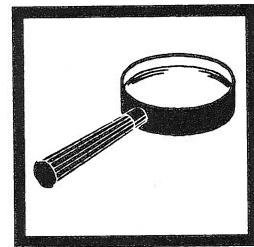
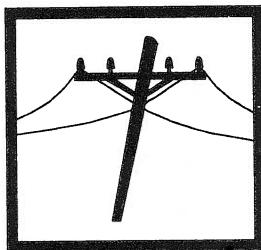
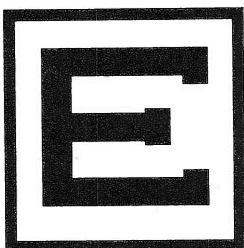
- Always make the lines and shapes at least 2 pels wide and 2 pels deep so that they will remain when converted to a lower density.
- Keep the designs simple with thick, bold lines and bright colors. Nothing catches the eye like a multicolored icon or bright mouse pointer. On the other hand, nothing degenerates faster when converted by PM than an image drawn with delicate, single-pel lines.
- Use only images and avoid text or letters. Where letters are used, stay with a single large, bold letter. Text quickly becomes unreadable when PM adjusts the density of the icon.
- Stay with primary colors in a mixed display environment. 8514 and XGA adapters support 256 colors, while EGA and VGA adapters support only 16 colors. Colors not supported by the lower-density adapters can present a problem.

- Consider the shape of the mouse pointer. It must have an instantly recognizable hot spot (see pointers later in this chapter).
- A custom mouse pointer should be as unique as possible as its appearance indicates to the user when your window has the focus. Selecting a pointer that is similar to the system pointer will confuse the user.
- A white border, or border that contrasts with the desktop, drawn around an icon helps it stand out from the background. White is a good selection because you cannot control the color of an individual user's desktop, and accidentally using the desktop's background color for the icon will cause that icon to become lost on a busy desktop.
- The user's desktop color is guaranteed to be different from your display's background color. This is important, because the background color is not a specific color but a notation to use the desktop color found on the user's display. So, if you do not distinguish the background screen color notation from the pallet color of the same shade, you may mistakenly use the pallet color when you really mean to specify the user's background color. To avoid this mistake, select *X-bkgnd* under the OPTIONS Action-bar item. This places an X in every grid box currently in the background color.
- Like the background color, your reverse color is almost guaranteed not to be the same on the user's display. Like the background color, the reverse color is a notation to PM, to use a color that contrasts with the user's background color. It is impossible to know what this color will be, so if colors are important to your object, stay away from reverse colors.

For detailed instructions on how to use this editor, see the *Presentation Manager Programming References*, part of the Developer's Toolkit.

Figures 8-1 and 8-2 show the icon developed for the sample program, drawn in a 32 x 32 pel size. Depending upon the display adapter that you are using, you may wish to adjust the density. You may use any colors that you wish in drawing the icon. For the sample program, I used black for the PM, blue for the COBOL and horizontal bars and white for the background. Feel free to experiment with your own colors. Remember, if you don't like your first try, you can quickly change it by re-editing the icon or mouse pointer, recompiling the resource, then binding the new resource to your program.

Here are three examples of icons I use on my system. They underscore some of the features I feel are important for icons. The E represents my editor. It is bold, very easy to spot and retains its image even when manipulated by PM. The telephone lines represents a communication program. The actual image uses several colors and, again, stands out well in the crowd at the bottom of the desktop. The magnifying glass represents a query program. No text is needed to identify the program represented by this icon.



Adding The Icon To The Resource Script File

The icon and mouse pointer resources are the first of the single line entries used in the Resource Script file. Up until now everything put into the Resource Script file has been multiple line table entries with a resource statement line, followed by paired BEGIN and END statements bracketing one or more resource definition lines. With single line entries, all the necessary data is contained on one source line. With icons and mouse pointers, the actual bit map object is contained in a separate file with only a reference to that file placed into the Resource Script file. During compilation the Resource compiler uses the resource statement to locate the icon or pointer object and incorporate it into the compiled resource file.

Here is the format to use when coding single line entries in the Resource Script file:

```
— resourcetype - resourceid — filename -
                                |           |
                                |           |
                                | loadoption | memoption
```

To define an icon within the Resource Script file, you first set the resource type to ICON. Place the resource identifier, the unique value that identifies this resource, as the first parameter. Since the icon is considered part of the frame window, use the same resource ID used to define the other Main frame window resources, the accelerator and menu tables.

Following the resource ID are optional load and memory parameters that define how the Presentation Manager should handle this object. These are the same load and memory options used with other resources. The load options, PRELOAD or LOADONCALL, define when the resource will be loaded into memory. The three memory options, FIXED, MOVEABLE and DISCARDABLE, indicate how much flexibility OS/2 has in managing these resources once they are loaded into memory. See Chapter 5 for a more detailed description of these load and memory options.

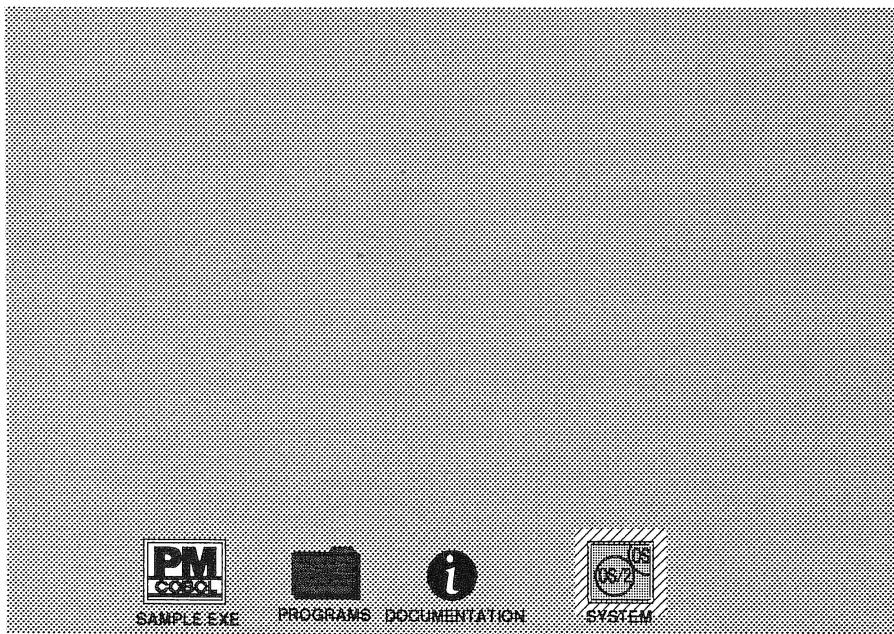


Figure 8-1 The Sample Program's icon (Version 2.0)

The last parameter is the name of the file that holds the icon or mouse pointer object. If the object file is not in the current subdirectory during the resource compile, then this must be the fully qualified path and file name of the object file. Here is the icon statement in the sample program's Resource Script file:

```
ICON ID_MainWind SAMPLE.ICO
```

By using an existing resource ID no additional entries are required in the Resource Header file, but a change is required to the COBOL program. A window icon is considered a frame window control, so PM must be informed that an icon is to be part of the main frame window by adding an additional Frame Creation flag. The Frame Creation flags were first set up in Chapter 4 when the frame window was built. The FCF-MENU flag was added in Chapter 5 when the Action-bar was added, and the FCF-ACCELTABLE was added in Chapter 7 when the first message box was created. Now the FCF-ICON flag must be added. As with the other Frame Creation flags, the binary sum of all the flags is the value passed to PM when the window is created. The flag FCF-ICON adds a value of 4000 hex to the existing flags, bringing the value of the variable *FCF-MAIN* to 0xca3f.

No actual code is required to cause the icon to appear when the window is minimized. Removing the window from the desktop and replacing it with the icon, and reversing the process when the icon is selected, is a function of the default PM Maximize and Minimize Control window processing. Returning the WM-MINMAXFRAME message to PM via the WinDefWindowProc call causes PM to automatically perform these functions.

Creating A Custom Mouse Pointer

As with icons, use the Icon Editor to create a custom mouse pointer. The only change required is selecting POINTER as the object type after opening a new object. The same reservations stated for icons apply to mouse pointers with the following addition. The hot spot within the mouse pointer is very important. The hot spot is the single pel within the mouse pointer that determines the exact point of sensitivity; that is, the exact point that locates the mouse pointer on the desktop. When the user selects an item with the

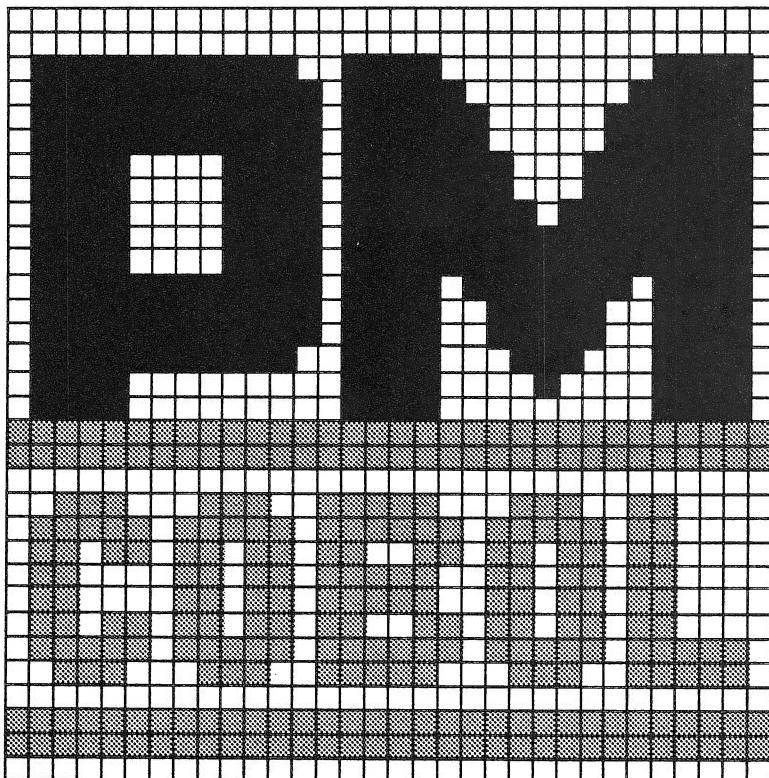


Figure 8-2 The Sample Program's icon design

mouse pointer, it is the location of the hot spot that actually determines what will be selected. To make that selection easier, the mouse pointer's hot spot should be located in such a place as to make it intuitively obvious to the user. With the system pointer it is the tip of the arrow, and with the sample program's hand pointer it is the tip of the index finger. If you create a pointer that hides the hot spot inside the image or choose an object without a natural point, the user will have trouble positioning the pointer to select controls that are in close proximity within the window. To better understand the importance of the hot spot, move the sample program's hand pointer slowly over the Main Frame window in a backward direction. Notice that the pointer does not change to the frame resize pointer (double-headed arrow) until the tip of the finger moves over the frame. Only then does PM recognize that the pointer is over the frame.

Adding The Mouse Pointer To The Resource Script File

The mouse pointer object, like the icon, is contained in a separate file until the actual compilation of the Resource Script file. As a result, the Resource Script file contains only a single line entry pointing to the mouse pointer object. The resource statement begins with the **POINTER** resource-type identifier. The first parameter is the resource ID. Unlike icons, a mouse pointer is not part of any window, but becomes a PM resource when loaded. As a result, the pointer can be shared by all windows within the process. To do this, it must have its own unique resource ID. The second and third parameters are the same load and memory options used with the icon. The final entry is the name of the file that contains the pointer object. If the mouse pointer object file is not in the current directory at the time of the resource compile, then this entry must be the fully qualified path and file name of the object file. Here is the pointer entry used in the sample program Resource Script file:

```
POINTER ID_Handptr HAND.PTR
```

Because the pointer's resource ID is new it must be defined in the Resource Header file and, as it will be referenced by the program, declared in the program's Working-Storage. Here are the two declarations used in the sample program, first the Resource Header file definition, then the program declaration.

```
#define ID_Handptr      901  
001820 77 ID-Handptr      pic 9(4) comp-5 value 901.
```

Adding The Pointer Code To The Sample Program

The mouse pointer is continuously displayed by PM. Every movement of the mouse causes PM to repaint the portion of the screen containing the mouse pointer to reflect its new position. If PM is not instructed to use a custom image, the default system pointer object will be used for each repainting of the pointer. To substitute a custom pointer, the program must tell PM to use the custom pointer every time the pointer is repainted as the result of movement.

Unlike an icon, utilizing a custom mouse pointer requires related code in the program. Four additional pieces of code are needed to support the display of a custom pointer. First, PM must load the mouse pointer from the Resource file into the system. Second, the WM-MOUSEMOVE message generated by the movement of the mouse, must be intercepted, so that the program knows when the mouse pointer is to be repainted. Third, the window must be tested to determine if it has the focus. A window without the focus must not display a custom pointer as the pointer indicates the window has the focus. Finally, the actual command to display the mouse pointer must be issued.

Coding The WinLoadPointer Call

The actual bit map that forms the pointer has already been created using the Icon Editor, and resides in a separate Resource file. The pointer object must be brought into memory and established as an available pointer. This is done with the WinLoadPointer command. Unlike the resources already discussed, pointers are not loaded into the program's Working-Storage. Pointers are considered system resources, displayed and manipulated by PM, and as such pointers are loaded from the resource file into system memory where they are kept under PM control. As you would expect with PM resources, once loaded, references to a pointer object are made via a handle rather than a resource id.

Loading the pointer via the WinLoadPointer call can be performed any time prior to the use of the pointer. The sample program will be using this pointer for the Main window's Client window, so the WinLoadPointer call must be made prior to the creation of the Main window. This ensures that upon receipt of the first WM-MOUSEMOVE message, the custom pointer can be displayed.

The WinLoadPointer call requires the label of the variable holding the handle of the desktop as the first parameter. The second parameter is the handle of the dynamic link library module holding the resource file that contains the pointer object. A null in this parameter indicates that the resource file is bound with the program rather than in a

separate dynamic link library. The third parameter is the variable holding the resource ID of the pointer. For the sample program this is the variable *ID-Handptr*. The final parameter is a standard signed long integer variable which will receive the handle of the pointer returned at the end of the call. As with all window elements, from this point on, the mouse pointer is identified via its handle. Here is the WinLoadPointer call used in the sample program:

```
005390      Call OS2API 'WinLoadPointer' using
00540              by value HWND-DESKTOP
00541              by value ShortNull
00542              by value ID-Handptr
00543          returning hwndPointer
```

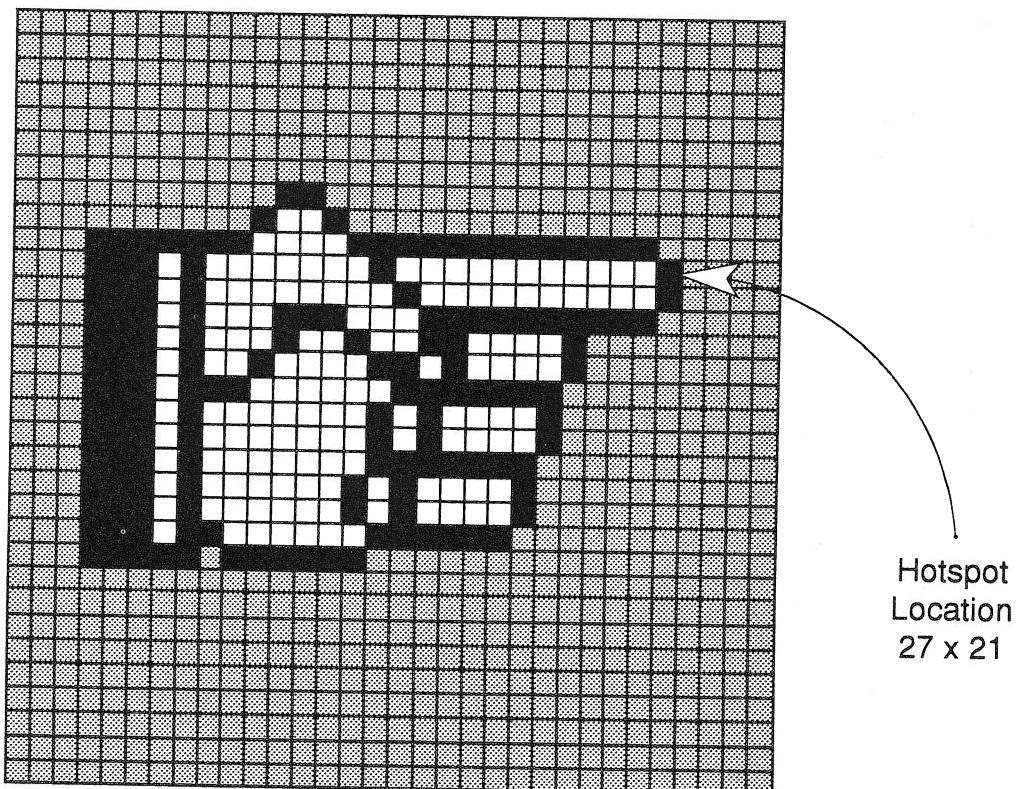


Figure 8-3 The Sample Program's hand pointer design

Checking For Movement Of The Mouse

The WM-MOUSEMOVE message is sent to a window whenever the mouse pointer moves across some part of its surface. The receipt of WM-MOUSEMOVE messages is not dependent upon the window having the focus. Anytime the mouse pointer moves across the surface of the window, the movement message is sent to the window. Remember, however, each window is actually several windows and the frame window and control windows, rather than the client window, receive the mouse movement messages when the mouse pointer moves across their surface. Each window that is to use the custom pointer object must intercept the mouse movement messages and instruct PM to use the custom pointer object. Since the sample program will initially display the custom pointer only within its Main window's Client window, only the WM-MOUSEMOVE messages sent to the MainWndProc are intercepted.

The WM-MOUSEMOVE message contains the current x and y coordinates of the pointer relative to the lower left-hand corner of the window. For the sample program the exact location of the pointer is not important, only the fact that the pointer moved and that it is currently located on top of the Main window's Client window. If, however, the pointer were moving among the controls of a dialog, the x y coordinates might be used to determine which control was to receive the focus.

To check for the mouse movement message, add the *when WM-MOUSEMOVE* phrase to the *Evaluate Msg* statement of the Main window procedure. When the WM-MOUSEMOVE message is intercepted, issue the WinSetPointer call to make the custom mouse pointer the current mouse pointer. This new pointer has already been loaded into memory, and the WinSetPointer call merely substitutes the new pointer's handle for the current pointer handle.

Coding The WinQueryFocus call

Intercepting the WM-MOUSEMOVE message indicates that the mouse has moved across the window, but not if the window has the focus. Remember, a program should not display a custom pointer unless the program's window has the focus. Changing the system pointer to the program's custom pointer indicates to the user that the window is ready to accept input, when in fact this may not be true.

To determine if this window has the focus, the WinQueryFocus call is issued to obtain the handle of the window with the focus. It is then a simple matter to compare the returned handle to the window's handle, and if they are equal, this window has the focus.

To code the WinQueryFocus call, place the variable holding the handle of the desktop in

parameter one. Parameter two is the window locked state and is no longer used by current releases of OS/2. So parameter two is coded as a null. The last parameter is a standard long integer that will receive the handle of the window with the focus. Here is the WinQueryFocus call used in the sample program:

```

009320      Call OS2API 'WinQueryFocus' using
009330          by value HWND-DESKTOP
009340          by value ShortNull
009350          returning hwndFocusWindow

```

Coding The WinSetPointer Call

Since there is only one mouse pointer and it is owned by PM, the WinSetPointer call requires the label of the variable holding the handle of the desktop as parameter one. Parameter two is the label of the variable holding the handle of the pointer that is to be made the current pointer (displayed). The last parameter is the label of the variable that is to receive the return code from the call.

Notice that when the handles are not equal, the message is not processed and must be returned to PM for default processing. Since the message has been selected by the *when WM-MOUSEMOVE* phrase, the normal default *when Other* phrase will not capture the message. So for WM-MOUSEMOVE messages when the Main window's Client window does not have the focus, a separate WinDefWindowProc call must be included as the Else phrase.

Here is the WinSetPointer call used in the sample program:

```

009370      If hwndFocusWindow = hwnd
009380          Call OS2API 'WinSetPointer' using
009390              by value HWND-DESKTOP
009400              by value hwndPointer
009410              returning ReturnData
009420      else
009430          Call OS2API 'WinDefWindowProc' using
009440              by value hwnd
009450              by value msg
009460              by value MsgParm1
009470              by value MsgParm2
009480              returning Mresult
009490      End-if

```

Chapter 9

Creating And Sending PM Messages

In prior chapters, I discussed how to intercept and process messages sent from other window procedures or from the Presentation Manager. To date, the sample program has intercepted and processed four different PM messages WM-COMMAND, WM-MOUSEMOVE, WM-PAINT and WM-QUIT. In this chapter, I will discuss how to construct and either send or post messages to other window procedures or to the Presentation Manager. The ability to send or post messages to another procedure allows one procedure to invoke functions within another procedure, and exercise control over the appearance and function of that procedure's window.

There are two ways that messages can be passed within the Presentation Manager system. Messages may be passed directly to another procedure by sending the message using the WinSendMsg call or the WinDispatchMsg call. Messages sent to another procedure are not processed through that procedure's message queue, but arrive directly at the procedure's entry point, bypassing the message processing loop and preempting any messages for that procedure that may be in the message queue.

Alternately, messages may be posted to another procedure's message queue using the WinPostMsg call or WinPostQueueMsg call. Posted messages are always placed into the target procedure's message queue and processed by the message processing loop on a first-in first-out basis. Because posted messages are placed into the message queue, there is usually some time delay between the posting of the message and its processing by the target procedure.

Both sent and posted messages have identical formats, and both types of messages are processed identically by the receiving procedure. The distinction between the two types of messages lies in the way control is transferred between the procedure generating the message and the procedure receiving the message. As almost any message may be posted or sent, understanding the difference in the transfer of control when sending a message versus posting a message is important.

Sending a message transfers control directly to the target procedure. In the same way that a standard COBOL call synchronously transfers control to the called procedure, sending a message suspends the sending procedure at the WinSendMsg call until control is returned by the target procedure. In fact, you should consider sending a message the same as calling a subroutine in COBOL, with one important difference. By using PM as the transfer mechanism and the standard message format as the data passing mechanism, a procedure sending a message needs no special understanding of the target procedure.

The WinDispatchMsg and WinSendMsg calls are identical in function and message format. The only difference between these two calls is the way the message is specified. The WinDispatchMsg call sends the message by passing a pointer to the message structure, while the WinSendMsg call contains the individual message parameters within the call.

The WinDispatchMsg call, used in every message processing routine, is a good example of sending messages. Messages removed from the message queue by the WinGetMsg call are passed directly to their target procedures by the WinDispatchMsg call, resulting in the suspension of the message processing loop until control is returned by the window procedure.

Messages should be sent when the sending procedure cannot continue execution until the receiving procedure has completely processed the message and returned the results to the sending procedure. But, just as you must understand the implications of calling a procedure, you must understand what actions the target procedure will take before sending a message to it. The continued execution of the sending procedure is dependent upon the prompt return of control by the target procedure.

Posting a message does not transfer control directly to the target procedure. The message is placed on the top of the target procedure's application message queue, and control is immediately returned to the posting procedure. The posting procedure continues executing without waiting for the message to be processed by the target procedure, or the results to be returned. The return code from the WinPostMsg call indicates only the success or failure of placing the message in the target message queue. Control eventually passes to the target procedure through normal OS/2 task switching. When control is obtained by the target procedure's thread, the message is processed through the standard message processing loop, eventually arriving at the target procedure's entry point.

The WinPostMsg and WinPostQueueMsg calls are identical in function and message format. The only difference between the two calls is that WinPostMsg places the message into the message queue of the window specified by the window handle in parameter one, while WinPostQueueMsg places the message into the message queue specified by the message queue handle in parameter one. Use of the WinPostQueueMsg call allows messages to be posted to procedures that do not support windows and thus do not have related window handles.

Posted messages should be used when the results of the message are not important to the procedure generating the message. That is to say, the posting procedure executes the same logic regardless of the actions taken by the target procedure.

Control Messages Within The Sample Program

To highlight the sending of messages, this chapter will add a window that will eventually display the source code of the PM calls used in the sample program. When the Source Code window is displayed, there is a potential conflict that is resolved through the sending of control messages to the menu. The sample program supports only a single copy of the Source Code window at any one time. To prevent the user from opening a second Source Code window, the menu entry that triggers the creation of the Source Code window must be disabled while the Source Code window is active, then enabled when the Source Code window is closed. This manipulation of the menu entry requires that messages be sent to the menu window, instructing it to disable or enable the Source menu entry. The message is sent rather than posted because the sample program cannot continue until acknowledgement that the menu item has been disabled or enabled is received.

Obtaining The Handle Of The Target Control Window

Before a message can be sent to a procedure, the entry point of that procedure must be passed to PM. Since individual window procedures are not defined within PM, a procedure must be identified through the window that it supports. Passing the handle of the supported window allows PM to identify the window class and thus determine the entry point of that class's procedure.

There are several PM calls that may be used to obtain the handle of a window. If the window was created by your program, then the window handle may be obtained based upon its relationship to the querying procedure's window (WinQueryWindow), its position within the window ancestry (WinWindowFromPoint) or through a process called window enumeration (WinGetNextWindow). See Chapter 13 for more information on window enumeration. If the window was not created by your program, as is the case with the menu window in the sample program, then the handle may be obtained based upon its relationship to a window created by your program (WinWindowFromID).

The menu control window was not created directly by the sample program, but it is the child to the Main Window's Frame window, so the WinWindowFromID call is used to obtain this handle. WinWindowFromID returns the handle of the control window with

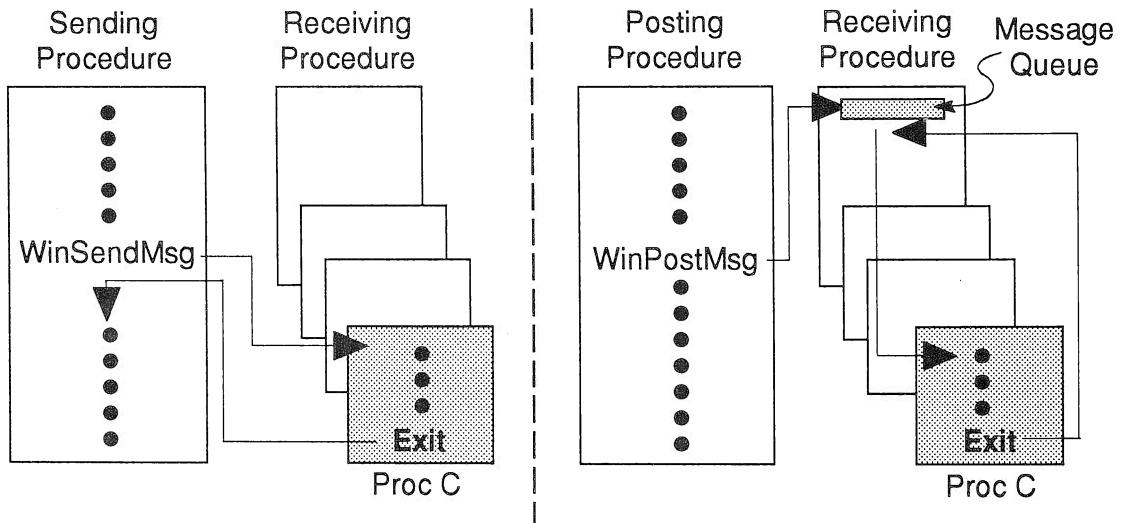


Figure 9-1 Sending a message vs. posting a message

the identity specified in parameter two. Each of the Frame window's control windows are known by a system-provided window identifier value called the Frame ID or FID. These values are used with the WinWindowFromID call to obtain the handle of a specific control window. For the sample program the value FID-MENU is used to obtain the menu window handle. Here is a list of the Frame IDs and their related values.

FRAME IDENTITY VALUES

FID-SYSMENU	The system menu, displayed as a space bar and located in the upper left-hand corner of the frame window.
FID-TITLEBAR	The application's title window, located at the top of the frame window.
FID-MINMAX	The maximize and minimize windows, displayed as upward and downward facing arrows and located in the upper right-hand corner of the frame window.
FID-MENU	The application's menu, running the width of the frame window and located just below the title bar.
FID-VERTSCROLL	The client window vertical scroll bar, running the height of the client window and located on the right-hand edge of the frame window.
FID-HORZSCROLL	The client window horizontal scroll bar, running the width of the client window and located at the bottom of the frame window.
FID-CLIENT	The frame window's application, or client, window.

To code the WinWindowFromID call, insert the variable holding the handle of the parent of the desired control window in parameter one. The parent of the menu control window is the Main Window's Frame window (hwndFrame). In parameter two, place the FID of the frame control window whose handle is to be returned. Parameter three is the standard signed, long integer variable that will receive the returned handle. Here is the WinWindowFromID call used to obtain the handle of the menu control window in the sample program:

```

007620      Call OS2API 'WinWindowFromID' using
007630              by value hwndFrame
007640              by value FID-MENU
007650          returning hwndMenu

```

Defining The Message Parameters

Every message contains two message parameters. Not every message uses both parameters, but both parameters must be built, including those defined as null. The contents of each predefined PM message is detailed in the *Presentation Manager Programming References*. These manuals contain separate chapters detailing the messages used with each control window. The chapter on menu control window processing details the menu messages (MM-) that are used to control the content and availability of menu items. You will recall that when the menus were first added to the sample program's Action-bar, their attributes were set to disabled status so the user could not select the individual menu entries before their supporting code was developed. By manipulating this attribute, the sample program can control the user's ability to select the Source menu entry and thus the user's ability to create the Source Code window. Specifically, the MM-SETITEMATTR message is used to set the attributes of the Source menu item. Here are the parameter values for use in the MM-SETITEMATTR message:

PARAMETER 1:

ITEM (UShort)	The menu item identifier
INCLUDESUBMENUS (Bool)	True - If the menu does not contain the menu item, search the submenu entries for the item. False - Do not search the submenus for the menu item.

PARAMETER 2:

ATTRIBUTEMASK (UShort)	The attribute mask - identifies which attributes to alter.
ATTRIBUTEDATA (UShort)	The attribute data - identifies how to alter the attributes specified in the mask.

Here is the message that must be sent to disable the Source menu entry, set the MIA-DISABLED attribute to true.

THE MM-SETITEMATTR MESSAGE					
Message	Id	MsgParm1w1	MsgParm1w2	MsgParm2w1	MsgParm2w2
(Hex)	0192	0084	0001	4000	4000
(Dec)	0402	0132	0001	16384	16384
MM-SETITEMATTR					
Menu Item (MI-SOURCE)					
True Flag					
Attribute Mask (MIA-DISABLED)					
Attribute Data (Set MIA-DISABLED to true)					

Note that the two UShort values in message parameter two are masks and not absolute values. The use of a bit-mask rather than a value allows a single message to alter all of a menu's attributes. The attribute data flag bits must be placed in the same relative position as the attribute mask. Thus, bit 7 of MsgParm2w2 must be set to true to activate the MIA-DISABLED attribute, bit 7 of MsgParm2w1. Because this message is dealing with only one attribute, both parameters can be set by simply copying the value 4000 hex to both parameters. When dealing with multiple attributes in one message, however, the mask and data flags must be individually set.

The message to enable the Source menu entry requires that the MIA-DISABLED attribute be turned off. This message is identical to the prior message with the exception of the attribute data (MsgParm2w2). The attribute data is set to false (0) to turn off the MIA-DISABLED attribute. As with the prior message, the sample program is dealing with only one attribute so the attribute data can be set by simply moving a 0 to parameter two. Here is the MM-SETITEMATTR message to enable the source menu entry:

THE MM-SETITEMATTR MESSAGE					
Message	Id	MsgParm1w1	MsgParm1w2	MsgParm2w1	MsgParm2w2
(Hex)	0192	0084	0001	4000	0000
(Dec)	0402	0132	0001	163840	0000
MM-SETITEMATTR					
Menu Item (MI-SOURCE)					
True Flag					
Attribute Mask (MIA-DISABLED)					
Attribute Data (Set MIA-DISABLED to false)					

Here is a list of the menu attributes that may be altered by the MM-SETITEMATTR message. These definitions are correct when true is set in MsgParm2w2. The reverse of these definitions is correct when MsgParm2w2 is set to false.

MENU ATTRIBUTES

MIA-NODISMIS	The menu containing this item is not hidden before notification is sent to the supporting code procedure.
MIA-FRAMED	Causes a frame to be drawn around the item.
MIA-CHECKED	Causes a check mark to appear next to the item.
MIA-DISABLED	Causes the item to be disabled.
MIA-HILITED	Causes the item to be displayed in reverse video.

Coding The WinSendMsg Call

With the message parameters built, the WinSendMsg call is issued to change the attribute of the Source menu entry. This call is coded by simply filling in the desired message parameters. Parameter one is the variable holding the handle of the window whose procedure is to receive this message. Parameter two is the message ID, and to change the menu attribute the MM-SETITEMATTR value is specified. Parameters three and four are the message's two parameters, built as four UShorts but sent as the two ULONGs, MsgParm1 and MsgParm2. The last parameter contains the variable that will receive the return code from the call. Remember, this is a sent message, so the sample program will be suspended until the frame window returns control along with a return code indicating the success or failure of the attribute alteration. Here is the WinSendMsg to disable the Source menu entry in the sample program:

```

011490      Move MI-Source to MsgParm1w1
011500      Move SetTrue to MsgParm1w2
011510      Move MIA-DISABLED to MsgParm2w1 MsgParm2w2
011520      Call OS2API 'WinSendMsg' using
011530          by value hwndMenu
011540          by value MM-SETITEMATTR
011550          by value MsgParm1
011560          by value MsgParm2
011570          returning ReturnData

```

The same code is used to reactivate the source menu item with the exception of line 014450 which sets only MsgParm2w1 to MIA-DISABLED (the same attribute is used), and line 014460 added to set MsgParm2w2 to false (0).

Updating The Resource Script File

To enable the Source menu entry from the View pull-down menu, remove the MIA_DISABLED menu attribute from the Source menu item together with the two preceding commas. To indicate that the accelerator keys CONTROL and the letter s key will invoke this source window, insert the \t alignment symbols followed by the character string Ctrl+s. Figure 9-2 shows the View pull-down menu with the addition of these accelerator keys to the Source menu entry. Here is the modified Source menu entry in the Resource Script file:

```
MENUITEM ">Source\tCtrl+s", MI_Source
```

To enable the accelerator keys for the Source menu entry, add a second entry to the table of accelerator keys defined for the *ID_MainWind* resource. Code this entry identically to the *MI_About* entry, changing the letter to s and the generated command value to *MI_Source*. To ensure that the program executes the same routine for this accelerator key combination as it does when the Source pull-down menu entry is selected by the mouse, use the same *MI-Source* command specified in the Main Window menu definitions.

Finally, a menu needs to be defined for the Source Code window. At a minimum, every window needs a clearly defined way for the user to close the window and exit the function. This could be done with a control push button, as in the About message box, or with a menu entry. If the menu is used, then the menu must also include a way for the user, having selected the pull-down menu, to escape the menu and return to the window without closing it. You should never place the user in the position of being forced to take an unwanted action just to escape the menu. The sample program's pull-down menu for the source window contains *Close* to close the window and end the function, and *Resume* to hide the pull-down menu and leave the source window unchanged.

I should add a word about the *Resume* menu item. As mentioned earlier, there must always be an escape to allow the user to return from a menu without taking any action. There are two ways to allow this menu escape. When the menu is displayed, selecting the same Action-bar entry a second time will cause the menu to be hidden with no action being taken, or a dummy entry can be added to the menu to give the user a selectable menu escape entry. For the sample program, I choose to add the dummy menu entry, *Resume*. By adding this menu entry and then ignoring the command generated when the item is selected, the menu is removed as part of PM's default message processing for the WM-COMMAND message. As a result, the sample program has a specific entry for the user that functions correctly, but does not require any code within the program.

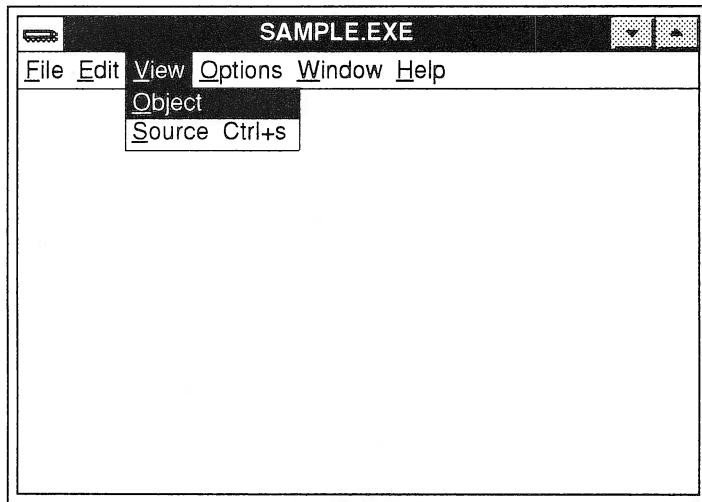


Figure 9-2 The View pull-down menu (Version 1.3)

This menu applies only to the Source window, so a new menu table must be created within the Resource Script file. As with the Main Window menu table, begin the table with a MENU statement defining this table as a menu resource. Include the resource ID that will relate this table to the child window. No optional load or memory options are specified as the default values are correct for this window. Following the MENU statement, add the SUBMENU and MENUITEM statements within paired BEGIN and END statements. Refer to Chapter 5 and Chapter 6 for more information on creating menu tables. Here is the Source Code window menu table:

```
MENU           ID_SourceWind
BEGIN
SUBMENU      "~Close",        ID_Close
BEGIN
MENUITEM    "C~lose",       MI_Closesw
MENUITEM    "~Resume",      MI_Resumesw
END
END
```

All resources created in this new menu table must be defined within the Resource Header file. Here are the required defines for the Source Code window's menu:

#define ID_SourceWind	20
#define ID_Close	210
#define MI_Closesw	211
#define MI_Resumesw	212

Don't forget to compile the resource file and bind the new version to the sample program. These changes will not be available to the program until the resource file is recompiled and bound to the program.

Adding The Resource File Changes To The Program

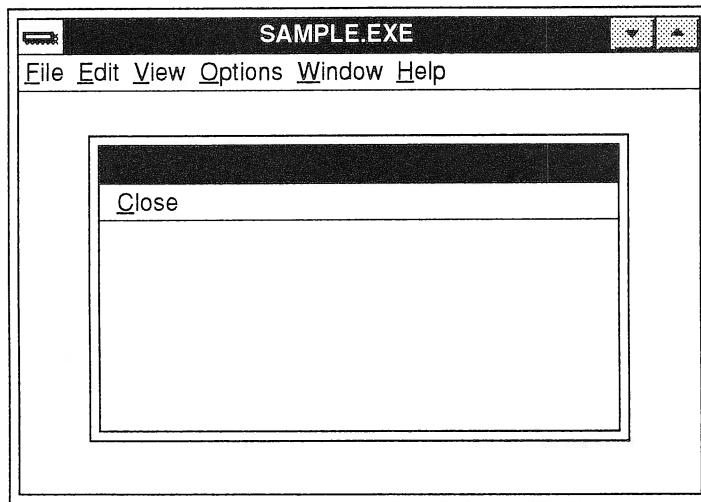
Any changes made to the Resource Header file that will be referenced by the program must be declared in the program's Working-Storage. MI-Resume is not defined as it is never referenced by the program. Remember, when transferring the definitions from the header file to Working-Storage, change any underscores (_) to dashes (-). Here are the required Working-Storage entries:

```
001550 77 MI-Source      pic 9(4) comp-5 value 132.  
001570 77 ID-SourceWind  pic 9(4) comp-5 value 20.  
001580 77 MI-Closesw    pic 9(4) comp-5 value 211.
```

Registering The Child Window

The steps for creating a child window are the same as for creating the Main Window: determine the class style to be used, set the class name, register the class, and issue the WinCreateStdWindow call. All of these functions have been coded before, so I will not detail them again. If you feel uncertain about coding any of these calls, review the creation of the Main Window in Chapter 4.

Regardless of when a window is created, the window should be registered in the Initialization routine. So the initialization routine must be changed to include the registration of the Source Code window. With the class style established and the class name declared, the window class supporting the Source Code window can be registered. Prior to the registration, the entry point for this class window procedure must be established using the COBOL Set statement. I code this statement just prior to the



**Figure 9-3 The Main Window with the Source Code
child window (Version 1.3)**

registration call because I use one procedure pointer variable for all registrations. The contents of the variable, *WindowProc*, needs to be valid only during the WinRegisterClass call. Once the entry point contained in this variable has been passed to PM, the content of *WindowProc* is no longer used. For information on coding the WinRegisterClass call, see Chapter 4. Here is the registration of the ChildWinClass used in the sample program:

```

005280      Set WindowProc to Entry 'ChildWndProc'.
005290      Call OS2API 'WinRegisterClass' using
005300          by value      hab
005310          by reference ChildWndClass
005320          by value      WindowProc
005330          by value      CSChildClass
005340          by value      UShortNull
005350          returning    RETURNDATA

```

Intercepting The MI-SOURCE Message

Now that source menu has been enabled, the routine to intercept and process this message must be added to the program. As with the *MI-About* command, the *MI-Source* command is passed to the program within a WM-COMMAND message. The routine to

extract and examine the contents of the WM-COMMAND message is already part of the sample program, so intercepting the *MI-Source* command only requires inserting another *IF End-IF* phrase into the *when WM-COMMAND* statement.

The *MI-Source* routine contains the same code used to create the Main Window, so I will not detail each call. The procedure consists of creating the window, querying the size of the Main window's Client window, calculating the size of the newly created window, and positioning and showing the new window. For a complete description of each of these calls, see Chapter 4.

The WinQueryWindowRect is coded differently in the *MI-Source* routine because the window being measured is the Main window's Client window. Knowing that all messages sent to the MainWndProc Section have the handle of the Main window's Client window, the query uses the handle taken directly from the message itself.

The code used to size and position the Source Code window is identical to that used with the Main Window. The calculations in this procedure create a window the same shape and only slightly smaller than the Main window's Client window. The WinSetWindowPos call is identical to the Main Window procedure. Remember, you can use any logic that you wish to set the size and position of your windows. The sample program's calculations are only examples.

Destroying Child Windows

By creating a new class of windows, a new code procedure is required to process messages sent to that class. As a minimum the entry point and the structure of the passed message must be defined along with the *evaluate Msg* statement and the *when other* phrase containing the WinDefWindowProc call. This is the minimum code that must be included in a window procedure, but for the sample program some of the existing window processing code used in the Main Window procedure has been duplicated to make the Source Code window more appealing. The routines to process the WM-MOUSEMOVE and WM-PAINT messages have been copied from the Main Window procedure. This ensures that the hand pointer appears within the Source Code window when it is displayed and that the Source Code window is properly repainted when required.

It is important to point out again that code procedures support window classes, not individual windows. Multiple windows can be supported by a single window procedure and the Source Code window could be supported by the *MainWndProc* if the only functions to be provided were the custom pointer, painting and the close processing. But a separate class was created because routines unique to the Source Code window will be added in later chapters.

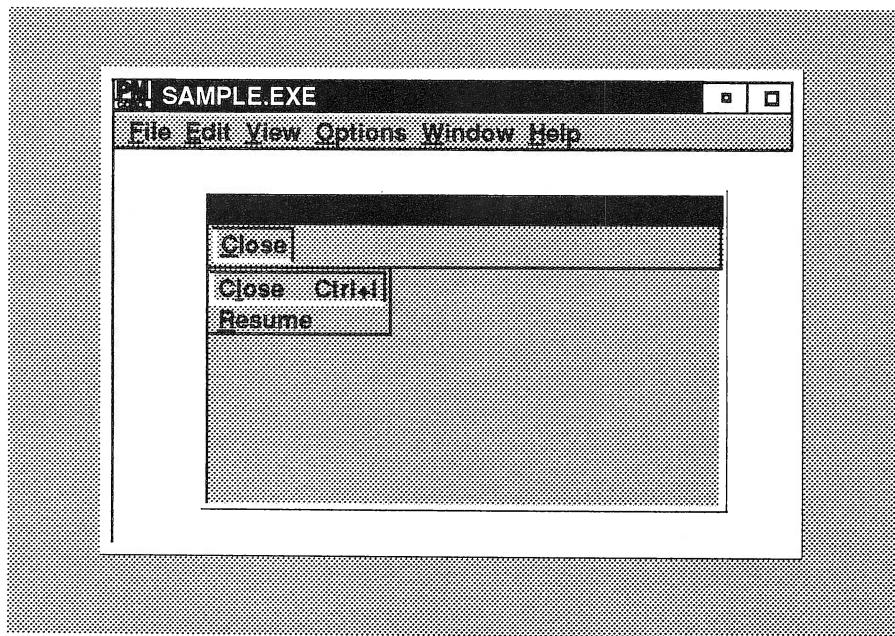


Figure 9-4 The Source Code child window with the Close pull-down menu (Version 2.0)

The only additional code needed in the *ChildWndProc* intercepts the Close menu command and destroys the Source Code window. As with all command messages, the *MI-Closesw* menu command is sent to the window procedure as a WM-COMMAND message. So to intercept the *MI-Closesw* command, add a *when WM-COMMAND* phrase to the COBOL Evaluate statement followed by a conditional *If MsgParm1w1 = MI-Closesw* statement. This routine contains the calls to destroy the child window and reactivate the Source menu item.

Destroying a child window is no different than destroying the Main Window. To destroy a window, issue the WinDestroyWindow call passing the frame window handle of the window to be destroyed as parameter one and coding the standard return code variable as parameter two.

Although the window destruction is identical, you should clear all destroyed window handle variables after the windows are destroyed. Handles are pointers that PM creates and maintains within your program's heap space. When a window is destroyed, the handle of that window and all of its child windows are removed from the heap space. However, the chances are very good that when another window is created, PM will use the same heap space for a new handle. If your program still has the old handles saved within its Working-Storage, a newly created handle may occupy the same heap space as

an old handle, causing an invalid match when checking for windows. To avoid this, always set the permanent handle variables to zero when the handle is no longer valid.

Finally, the Source menu entry must be enabled by sending a MM-SETITEMATTR message to the menu, specifying that the MIA-DISABLED flag be turned off. There is no enable flag, so menu items are controlled by turning the MIA-DISABLED flag on and off. See the prior discussion of the MM-SETITEMATTR message for the exact format of this message.

Chapter 10

Working With Dialogs

Up to this point I have concentrated on the structure of Presentation Manager programs, sending and receiving Presentation Manager messages and creating various windows. Now I will begin discussing the process of collecting user input. For PM programs, collecting user input means using dialogs and dialog boxes. In Chapter 7, I introduced user dialogs using a very specialized form of dialog box, the message box. In this chapter, I will begin the discussion of generalized dialogs and dialog boxes with the creation of a program termination dialog.

Dialog boxes are specialized windows used by an application whenever a conversation with the user is required. This conversation can be one-way (collecting input or issuing a message) or two-way (collecting data and returning requested information). Dialog boxes can contain any combination of system-supplied and application-defined control windows in any arrangement the programmer desires. Both types of control windows are treated as child windows of the dialog box, with the dialog box itself acting as a unifying mechanism for what can be a complex arrangement of control windows.

Because the dialog box owns the control windows, each individual control window procedure sends a notification message to the dialog box when a significant event occurs to a control window. Conversely, when the dialog box wants to alter or query one of its control windows, it sends a control message to the control window procedure. The application's dialog procedure in turn, talks to the dialog box through a series of notification and control messages instructing the dialog box how to manage the individual control windows.

Modal vs. Modeless Dialog Boxes

All dialog boxes are either modal or modeless in their operation.

A modal dialog box prevents the user from interacting with any other part of the application while it is active. The user may not interact with the current window or shift the focus to any other window within the application. Until the program dismisses the dialog box, the user can only interact with the dialog box. This is an important feature because it ensures that the user will respond to the program's request for information before proceeding with other program functions.

There are two types of modal dialog boxes, application modal and system modal. An application modal dialog box prevents the user from interacting with any other part of the application, but the user is free to shift to other applications while the dialog box is active. A system modal dialog box prevents the user from interacting with any other process within the OS/2 system. This includes all PM applications as well as applications running in other sessions. Because system modal dialogs have the power to effectively lock the system, they should be used with great care. System modal dialogs are usually limited to events that require the user to react before any other event within the system can occur, such as an unrecoverable system error.

A modeless dialog box does not prevent the user from accessing the application that created the dialog or other applications. This mode of operation is designed primarily to let the user select information rather than input information. This type of dialog box usually remains active for long periods of time, allowing other functions to be performed while it is active. A graphical drawing program's toolbox is a good example of a modeless dialog box. It remains on the screen and may be selected at any time but does not interfere with the operations of the drawing program.

The modal or modeless nature of a dialog box is established at the time it is created. The dialog box will be modal if the WinDlgBox Presentation Manager call is used, or if the WinLoadDlg call or WinCreateDlg call is used followed by a WinProcessDlg call. The dialog box will be modeless if the WinLoadDlg or WinCreateDlg Presentation Manager call is used and is not followed by a WinProcessDlg call.

The Program Termination Dialog Box

This chapter will add a dialog to the sample program to warn the user that the sample program is about to end and ask for verification that the program may end. This dialog requires the user to select the *OK* push button to end the sample program, or select the *No* push button to cancel the request to end the program and allow the user to continue using the sample program. See Figures 10-1 and 10-2 for examples of the Program Termination Dialog box.

This is about as simple a dialog as can be created, but like the message box that was added to the sample program earlier, it utilizes all of the basic features of every dialog and lays

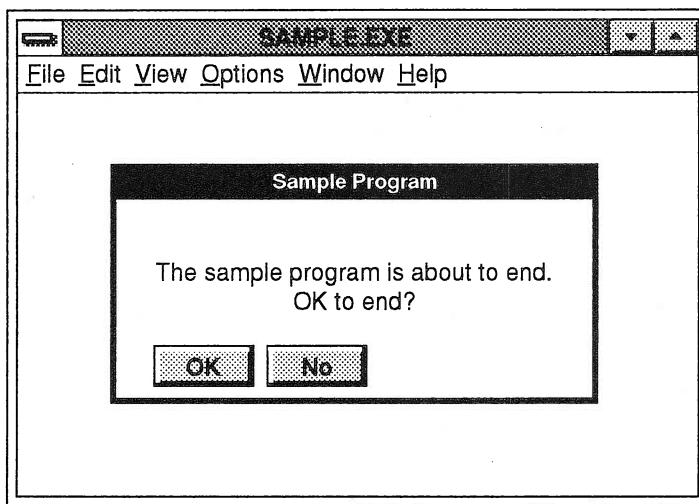


Figure 10-1 The Main Window with the Program Termination dialog box (Version 1.3)

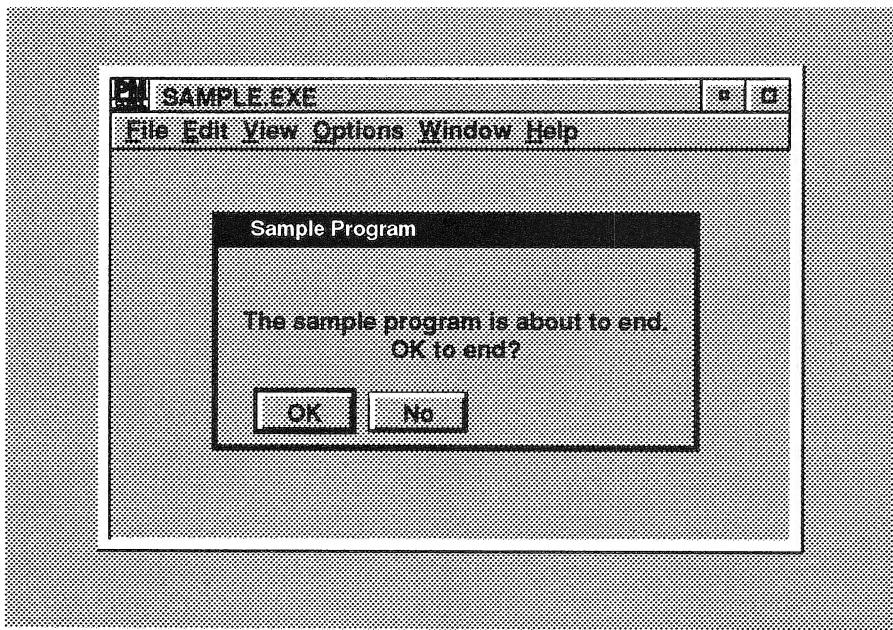


Figure 10-2 The Main Window with the Program Termination dialog box (Version 2.0)

the foundation for more complex dialog processing in future chapters. By keeping this dialog simple, you can more easily see the structure of dialog processing. To implement the Program Termination dialog, you will be required to:

- Construct the dialog box using a dialog template.
- Add the dialog box to the resource file.
- Issue the PM calls to create and dismiss the dialog.
- Code the dialog procedure.

Creating The Dialog Box Template

All dialogs are built from a series of specifications called the dialog box template. There are two ways that a dialog box template can be created. A dialog box template can be created prior to the program's execution and stored as a resource. Dialog boxes built prior to execution of the program that uses it are called static dialog boxes. Static dialog boxes are not as inflexible as their name implies. Static dialogs can be, and usually are, modified at execution time. Indeed, any control window that contains variable data is filled after the dialog box is created. If required, additional *user controls*, may be added to a static dialog box after it is built.

Alternately, a dialog box template can be created by the program during execution, temporarily stored in memory, then passed to PM at the time the dialog box is required. Dialog boxes built in the manner are called dynamic dialog boxes. They offer the program the maximum amount of freedom to adjust the contents of the dialog box to fit the current environment.

For the sample program, I have implemented static dialog boxes; that is, boxes created prior to the program's execution using the Dialog Box Editor. These are the most common types of dialog boxes and are less complex to implement than dynamic dialog boxes.

Using The Dialog Box Editor

Static dialog box templates are constructed using the Dialog Box Editor. The Dialog Box Editor allows the developer to draw the dialog box on the screen in the form that the user will see, and then converts the drawing into the specifications PM requires to build the box during execution. Information on the use of the Dialog Box Editor can be found in the *Presentation Manager Programming References*, part of the Developer's Toolkit. I will not attempt to teach the use of the Dialog Box Editor, but there are several hints that will make using the editor much easier.

- Be sure to spend some time practicing with the editor. Not all of its functions are obvious, and you should plan on making several false starts before you become familiar with the editor.
- You should have a good idea about the layout of the dialog box and its control windows before you start building the actual template. Knowing the number of control windows, their type and location before using the editor will help immensely.
- Before drawing the dialog box, define all the Resource identity values along with their labels in a dialog header file. To do this, you will need the layout of the dialog box that is to be created. If you don't have the labels predefined, you can easily become confused during the drawing and inadvertently assign the wrong resource value to a control window.
- Be sure to define all style information and memory options for the dialog box before you begin drawing the control boxes. Some styles, such as a title bar, are not generated automatically, and you will need to pay close attention to be sure to generate the styles you want. If the template is to be stored as a dynamic link library, be sure to specify the required load and memory options.
- Edit the dialog template file produced by the Dialog Box Editor. The output of the editor is sloppy, and some simple rearranging of the output can make this file significantly easier to read. The dialog file is a standard ASCII text file and your favorite editor should be able to make the changes that you require. See Figure 10-4 for an example of how to rearrange a dialog file. During the editing of the

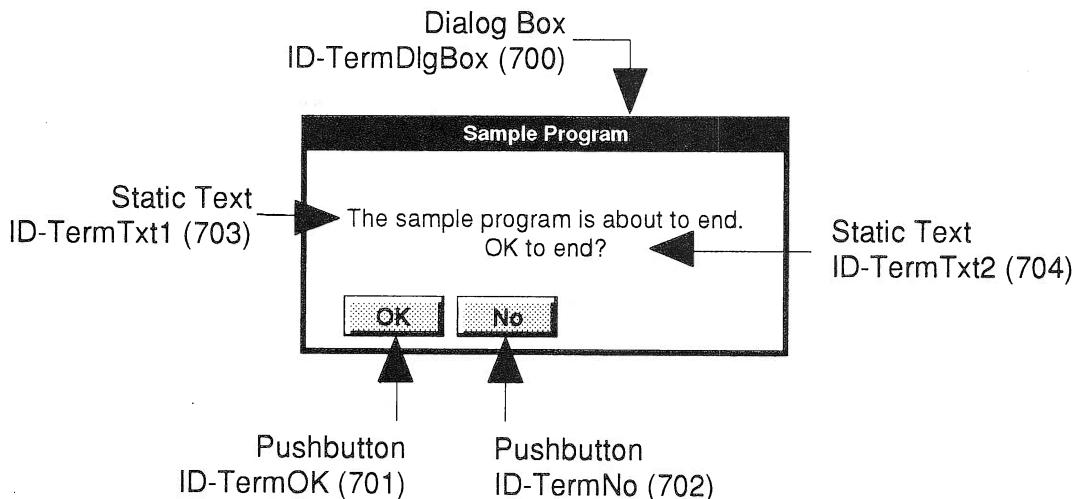


Figure 10-3 Resources for the Program Termination dialog box

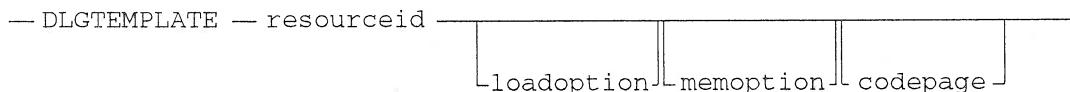
dialog template file, take out the DLGINCLUDE reference to the dialog header file. Removing this statement allows the dialog header file to be referenced using the standard #include statement at the beginning of the Resource Script file. Referencing all of the header files together at the beginning of the Resource Script file make for better documentation.

- Dialogs can also be modified using the Dialog Box Editor. But I find it infinitely easier to simply edit the ASCII dialog file and make my changes. I think you will find this to be the case too and I encourage you to practice making changes to the ASCII dialog file using a simple dialog template. Try moving the location of the control windows, changing the text, or altering the style of the dialog box. These changes should help you feel more comfortable with the content of the ASCII dialog template.
- Don't embed the dialog template directly into the Resource Script file. Their inclusion will diminish the readability of the Resource Script file. Instead, add a reference to the dialog template at the end of the Resource Script file using the resource directive RCINCLUDE.

Interpreting The Dialog Template

The dialog template created by the Dialog Box Editor is, to say the least, somewhat confusing. However, the content of a dialog template is straightforward and a little rearranging of the file will make the template very readable.

As with the other Resource Script file entries you have built, the dialog template begins with a Dialog Template Statement (DLGTEMPLATE) identifying this resource as a dialog template, assigning a Resource ID to the dialog and fixing any load, memory or codepage options that apply. Here is the format of the Dialog Template Statement:



The **resourceid** is the numeric value that becomes the identifier of the dialog resource. Each dialog resource is known to PM by this numeric value and when referencing a dialog prior to its being built, you must use the resource identity value.

The **loadoption** defines to OS/2 when the resource should be loaded. PRELOAD indicates that the module should be loaded immediately, in other words at the start of

```

DLGTEMPLATE ID_TermDlgBox
BEGIN
    DIALOG "Sample Program", ID_TermDlgBox, 66,42,204,57,
        WS_CLIPSIBLINGS | WS_SAVEBITS
        FCF_TITLEBAR | FCF_NOBYTEALIGN | FCF_DLGBORDER
    BEGIN
        CONTROL "OK", ID_TermOK, 14,5,38,13, WC_BUTTON, BS_PUSHBUTTON |
            BS_DEFAULT | WS_TABSTOP | WS_VISIBLE
        CONTROL "No", ID_TermNo, 62,5,38,13, WC_BUTTON, BS_PUSHBUTTON |
            WS_TABSTOP | WS_VISIBLE
        CONTROL "The Sample program is about to end.", ID_TermTxt1, 1,35,200,8,
            WC_STATIC, SS_TEXT | DT_CENTER | DT_VCENTER | WS_GROUP |
            WS_VISIBLE
        CONTROL "OK to end?", ID_TermTxt2, 1,25,200,8, WC_STATIC, SS_TEXT |
            DT_CENTER | DT_VCENTER | WS_GROUP | WS_VISIBLE
    END
END

```

```

DLGTEMPLATE ID_TermDlgBox
BEGIN
    DIALOG "Sample Program", ID_TermDlgBox,
        66, 42, 204, 57,
        WS_CLIPSIBLINGS | WS_SAVEBITS
        FCF_TITLEBAR | FCF_NOBYTEALIGN |
        FCF_DLGBORDER
    BEGIN
        CONTROL "OK", ID_TermOK,
            14, 5, 38, 13,
            WC_BUTTON,
            BS_PUSHBUTTON | BS_DEFAULT | WS_TABSTOP |
            WS_VISIBLE
        CONTROL "No", ID_TermNo,
            62, 5, 38, 13,
            WC_BUTTON,
            BS_PUSHBUTTON | WS_TABSTOP | WS_VISIBLE
        CONTROL "The sample program is about to end", ID_TermTxt1,
            1, 35, 200, 8,
            WC_STATIC,
            SS_TEXT | DT_CENTER | DT_VCENTER |
            WS_GROUP | WS_VISIBLE
        CONTROL "OK to end?", ID_TermTxt2,
            1, 25, 200, 8,
            WC_STATIC,
            SS_TEXT | DT_CENTER | DT_VCENTER |
            WS_GROUP | WS_VISIBLE
    END
END

```

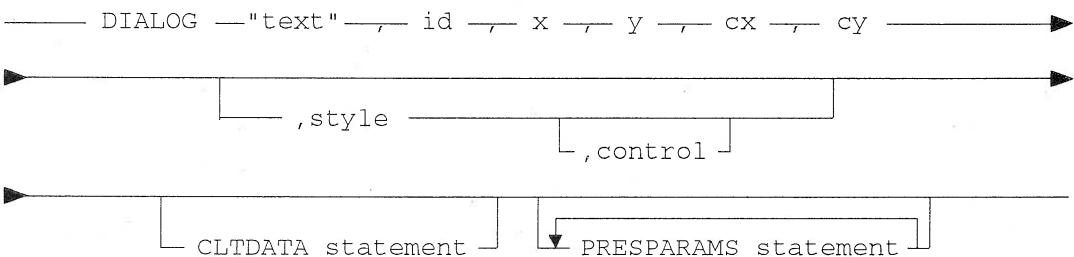
Figure 10-4 The Program Termination dialog before and after editing

program execution. `LOADONCALL` indicates that the resource should be loaded only when it is called by the application. The load options are mutually exclusive.

The **memoption** define how much flexibility OS/2 has in managing the resources once they have been loaded into memory. FIXED indicates that, once loaded, the resources must remain at a fixed memory location. The MOVEABLE memory option indicates that the resource module may be moved by OS/2 if memory is being compacted. The DISCARDABLE memory option indicates that the resource may be discarded when its memory is required or it is no longer needed.

The **codepage** identifies the table to be used to translate the ASCII characters into the dialog's text. If no codepage is specified here, the codepage specified for OS/2 will be used.

Following the Dialog Template Statement there must be at least one DIALOG statement. The Dialog statement identifies the beginning of the actual dialog and supplies the dialog's resource identity, specifies the size and location of the dialog, assigns the frame and style values to the dialog, and adds an optional dialog title. Since the dialog template contains multiple statements, the Dialog statements within the template must be placed within paired BEGIN and END statements. Here is the format of the Dialog statement:



The **text** is the text string used as the title of the dialog box. The title is optional.

The **id** is the resource identifier value to be assigned to this dialog box.

The **x** and **y** coordinates mark the lower left corner of the dialog box. The **x** and **y** values are specified in dialog coordinates and not window coordinates. See the following paragraph on the differences between dialog coordinates and window coordinates. For normal dialogs, these coordinates are relative to the origin of the parent window. For dialogs with the FCF-SCREENALIGN style, these coordinates are relative to the origin of the display screen. For dialogs with the FCF-MOUSEALIGN style, these coordinates are relative to the position of the pointer at the time the dialog is created.

The **cx** and **cy** coordinates set the width and height of the dialog box. As with the x and

y coordinates, the cx and cy values are specified in dialog coordinates, not window coordinates.

The **style** values represent additional frame and class specific styles to be applied to the dialog.

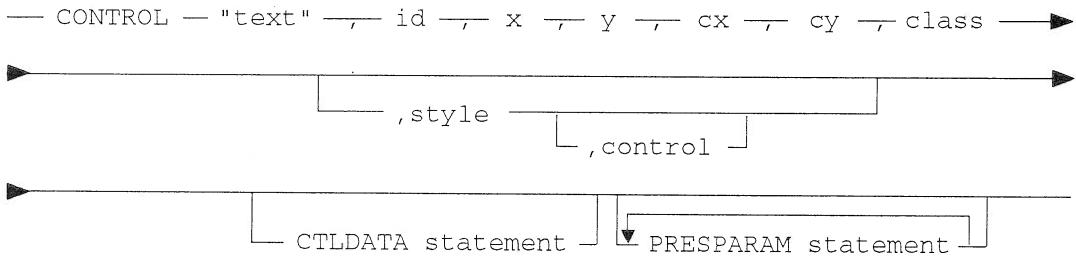
The **control** values are the Frame Creation flags to be applied to the dialog box.

The optional **CTL DATA** statement is used to define control data for the dialog control.

The optional presentation parameters phrase is used to change the colors, font face or font sizes of text and controls within this dialog. The PRESPARAM phrase is composed of an attribute type paired with a value for the attribute. There may be as many sets of attributes and values as necessary. Here is the structure of the PRESPARAM statement. See Appendix D for a complete list of presentation parameter attributes.



Individual controls within the dialog box are defined using CONTROL statements. Since a dialog is composed of multiple Control statements, all the Control statements for a single Dialog statement must be within paired Begin and End statements. With the exception of the class entry, Control statements are identical to Dialog statements in format. Here is the format of the Control statement:

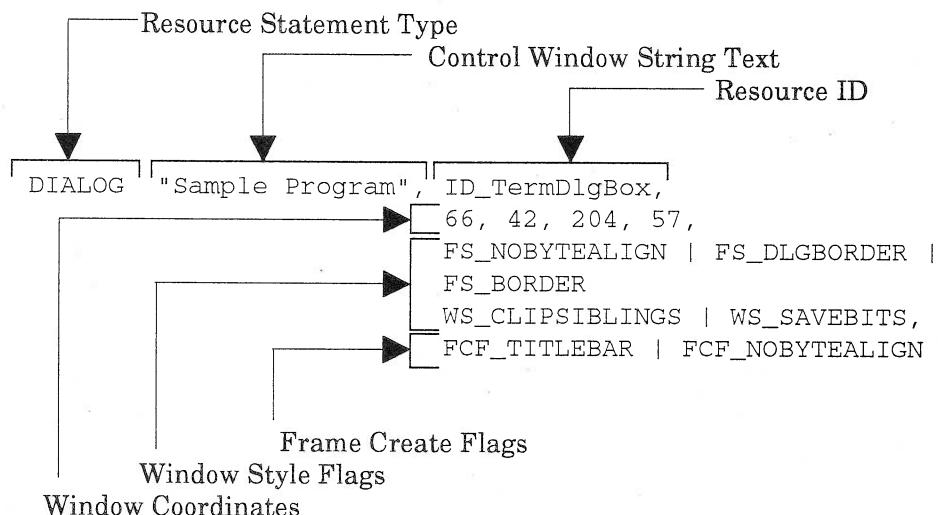


The **class** statement sets the window class of this control window. Any of the standard WC- values may be specified, one per control statement.

A word about the dialog box window coordinates is in order. These are not the window coordinates that you have used in conjunction with the PM window calls. Dialog coordinates are unique to dialog boxes and are based on the default character-cell size.

In the horizontal direction (X axis) one dialog unit is equal to 1/4 the default character-cell width, and in the vertical direction (Y axis) one dialog unit is equal to 1/8 the default character-cell height. Using dialog coordinates, the current character-cell is defined as 4 dialog units wide and 8 dialog units high. Translating the dialog coordinates, the Program Terminate dialog box has a width of 51 character-cells (204/4) and a height of just over 7 character-cells (57/8). This may seem to be a small dialog box, but character-cells include the leading space as well as the individual characters resulting in a larger space than that occupied by the visible character. Defining the dialog box in terms of the current characters ensures that the dialog box will retain its shape and capacity regardless of the point size of the characters.

To better understand the content of any of the dialog template statements, here is the Dialog statement used in the Program Termination dialog. Note that multiple style and control values are ORed together using the C language OR operator, the single vertical bar (|). For a more complete description of the style and control flags, see Chapter 1. The complete Program Termination dialog source is shown in Figure 10-4.



The Generated Dialog Template Format

Before PM can build a dialog box a compiled dialog template must be created using the ASCII dialog template as input. The dialog template is generated by the Resource compiler using the ASCII dialog template and dialog header file as input. The ASCII dialog template is the source code used by the Resource compiler to create the compiled dialog template. The compiled dialog template is composed of three sections

(see Figure 10-5). For a complete description of the template format, see the *Presentation Manager Programming References*.

Header. Defines the type of template format and contains information about the location of the other sections of the template.

Items. Defines each of the controls that comprise the dialog box, specified as elements of an array. This array also defines the hierarchy of the control windows within the dialog box. Each element has a window descriptor and pointers to the three data areas.

Data Areas. Contains the data values associated with each control. The data area also contains presentation parameter definitions.

Updating The Resource Script File And The Application

With the ASCII dialog template created as a separate file, the Resource Script file must be updated to allow the Resource compiler to find the source and build the compiled template. This modification requires only two changes to the Resource Script file, the #include statement referencing the resource definitions in the dialog header file, and an RCINCLUDE statement, pointing to the ASCII dialog template. See the Resource Script file in Appendix A, for the exact placement of these two statements. Remember, I am assuming that the DLGINCLUDE statement has been removed from the ASCII dialog template. If it hasn't, leave out the #include statement referencing the TermDlg.h file.

Any time resource definitions are added to the Resource Script file and are referenced by the program, duplicate declarations must be made in the program's Working-Storage section. As with other transfers from the header files, don't forget to change all underscore (_) characters to dash (-) characters. Here are the required COBOL declarations for the Program Termination dialog box resources.

```
001600 77 ID-TermDlgBox      pic 9(4) comp-5 value 700.  
001610 77 ID-TermOK        pic 9(4) comp-5 value 701.
```

In addition to inserting the dialog template into the Resource Script file, you will want to activate the *Exit* entry from the *File* pull-down menu. Activating this menu entry will give the user a program termination option that is part of the application, so that the System Menu is not required to end the program. To activate the *Exit* entry, remove the MIA_DISABLED and two proceeding commas from the *Exit* MENUITEM.

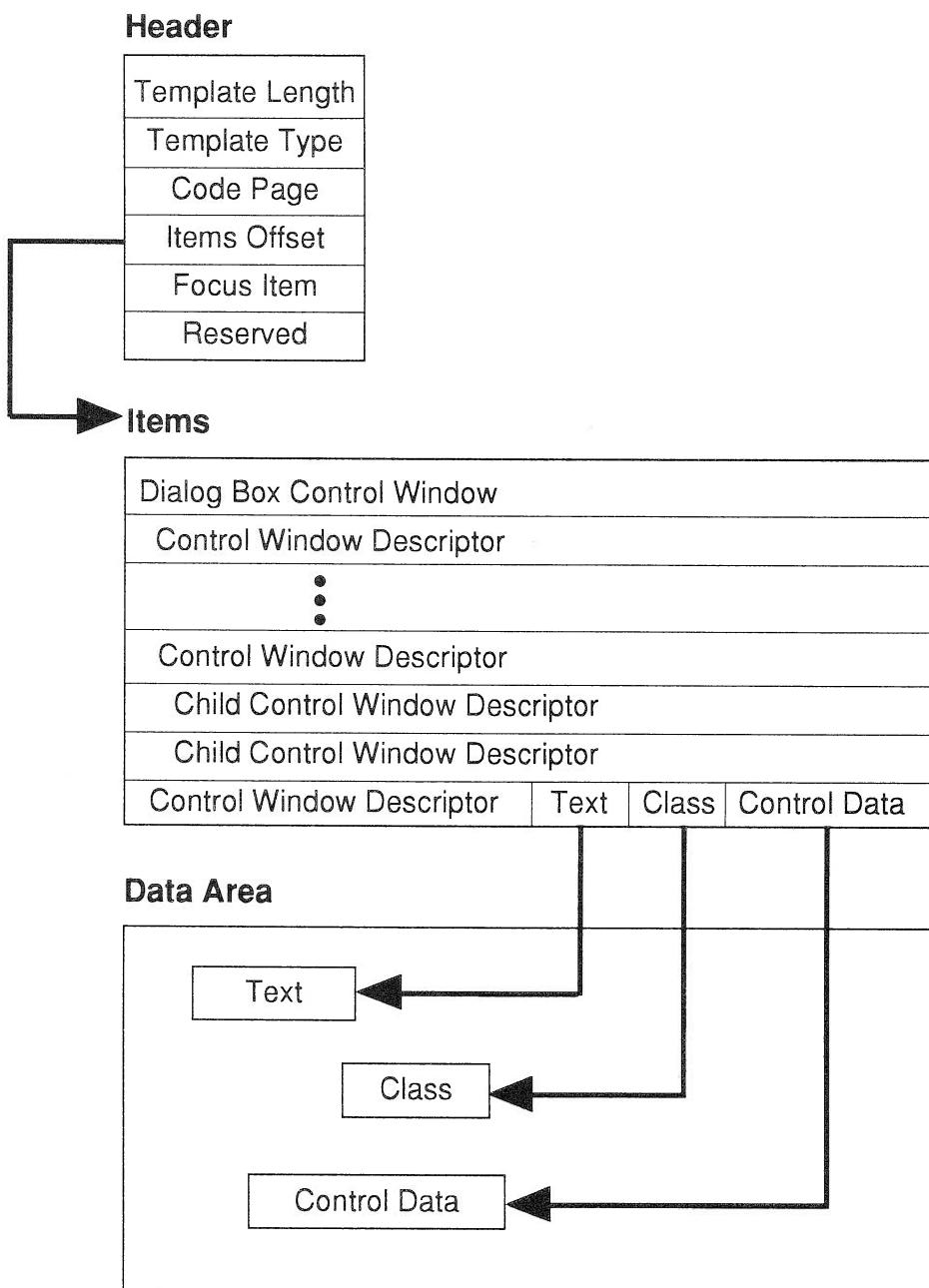


Figure 10-5 The compiled dialog template structure

After modifying the Resource Script file, remember to rebuild the Resource file using the Resource compiler, then bind the new Resource file with the modified sample program.

Generating The WM-QUIT Message Internally

The sample program termination dialog will be issued to verify that the program should end whenever the user indicates program completion. The user signals this by selecting a control that generates the WM-QUIT message. Up until now the only control that generates the WM-QUIT message is the *Close* entry within the System Menu. In the sample program, however, a close or exit function will be offered as part of the main menu.

With the program termination routine already in place for the WM-QUIT message generated by the System Menu, the sample program needs only to post a WM-QUIT message to itself whenever the user selects the program's *Exit* menu item. This ensures that a single program termination routine will support all requests to shutdown the program regardless of how the user signals the end of the program.

Since a WM-COMMAND message will be generated whenever the *Exit* menu item is selected, a new menu command routine must be added to the WM-COMMAND message routine in the MainWndProc Section. This section will intercept the MI-Exit message and issue the WinPostMsg to *hwndFrame*, causing the message to be placed into the application's Message Queue.

Coding The WinPostMsg Call

The WinPostMsg call is coded exactly the same as the WinSendMsg call added to the sample program in the last chapter. The WinPostMsg call requires the four parameters of the message to be sent plus a variable to receive the return code from the call. Remember, the only difference between sending and posting a message is how control is transferred between the two processes; the structure of the message is always the same. When the message is posted, control returns to the message-generating process as soon as the message is placed in the receiving process's message queue.

The first parameter is the name of the variable that holds the handle of a window that identifies the message queue to receive this message. The Main window's Frame window represents the procedure that uses the desired message queue, so this parameter is coded as *hwndFrame*. Parameter two is the message ID, WM-QUIT. Parameters three and four are the two message parameters. Neither is used for the WM-QUIT message, so both

must be set to null values. The last parameter is the variable to receive the return code. Remember that this return code only indicates the success of the posting function, not the completion of the message as with the WinSendMsg call. Here are the *MI-Exit* interception and the WinPostMsg call used in the sample program.

```

009790      If MsgParm1w1 = MI-Exit
009830          Call OS2API 'WinPostMsg' using
009840              by value hwndFrame
009850              by value WM-QUIT
009860              by value LongNull
009870              by value LongNull
009880          returning ReturnData
009900      End-if

```

Coding The WinDlgBox Call

This dialog is activated when the WM-QUIT message is received, so the WinDlgBox call, to initiate the dialog, is placed as the first function of the WM-QUIT message processing routine.

The WinDlgBox call locates the dialog template, loads, builds and displays the Program Termination dialog box as an application modal dialog. In addition, WinDlgBox suspends the WM-QUIT routine and transfers control to the Program Termination dialog procedure at the address passed as the third parameter of the call. The WM-QUIT routine remains suspended until the dialog procedure returns control by issuing the WinDismissDlg call.

When control returns, the return code must carry sufficient information to allow the calling routine to determine the success or failure of the dialog. This code may be any value that has meaning to the program, but I have found that passing the resource ID of the button, or other control, selected by the user to end the dialog is usually the best value to return. For the Program Termination dialog, the resource ID of the button selected by the user is returned as the WinDlgBox return code. The *MI-Exit* routine simply checks if the *OK* button was selected and, if true, sets the termination flag *program-done* to true. Receipt of the *No* button prevents the termination flag from being set.

To code the WinDlgBox call, place the variable holding the handle of the dialog's parent in parameter one. Any window handle may be specified, but as with the message box,

you run the risk of having that window closed or too small to hold the dialog box when it is created, thus clipping the dialog box and preventing the user from seeing its contents. By specifying the desktop as the parent, you are assured of having the dialog box displayed. To see the advantage of using the desktop as the parent of the dialog, minimize the sample program and select *shutdown* from the desktop.

For parameter two, specify the handle of the owner of the Dialog box. Specifying the owner of a dialog is somewhat less important for the dialog than for other windows because PM will calculate the actual dialog box owner at execution. The dialog box owner has no effect on the flow of normal messages as it does with windows, because dialog messages always flow to the procedure set in parameter three. As a result, I usually specify the handle of the window creating the dialog as the parent. For this dialog, the Main window's Frame window is creating the dialog, so *hWndFrame* is coded as parameter two. Parameter three contains the entry point address of the dialog procedure. The variable used in parameter three must be declared as a procedure-pointer. In addition, this variable must be loaded prior to the call using the COBOL Set statement. If the compiled dialog template is being held in a dynamic link library, parameter four is used to pass the name of the dynamic link library module to PM. Code parameter four as a null to indicate that the resource is bound with the program. Parameter five is coded with the variable holding the dialog's resource identity. This is the value established in the Dialog statement.

Parameter six allows the main procedure to establish a data area that can be used to pass initialization data to the dialog procedure and allow the dialog procedure to return data to the calling procedure. Parameter six contains the address of an area in the COBOL program's data section formatted according to the CREATEPARAMS structure. This structure has the following format:

LENGTH (UShort)	Length of the structure in bytes
TYPE (UShort)	Reserved
DATA (Bytes)	Application defined data area

The program termination dialog has no data to return, so this parameter is coded as a null address (LongNull). But, dialogs introduced in later chapters will utilize this dialog data passing area to initialize dialog variables and return user data to the window procedure. The last parameter is the variable that will receive the dialog's return code. This code is generated by the dialog procedure and passed to the procedure by PM. For the program termination routine, this is the resource ID of the button selected by the user. Here is the code to load the dialog's entry point address and the WinDlgBox call used in the sample program:

```
007840      set WindowProc to ENTRY 'TermDlgProc'
007850      Call OS2API 'WinDlgBox' using
007860          by value HWND-DESKTOP
007870          by value hwndFrame
007880          by value WindowProc
007890          by value ShortNull
007900          by value ID-TermDlgBox
007910          by value LongNull
007920      returning DlgReturn
```

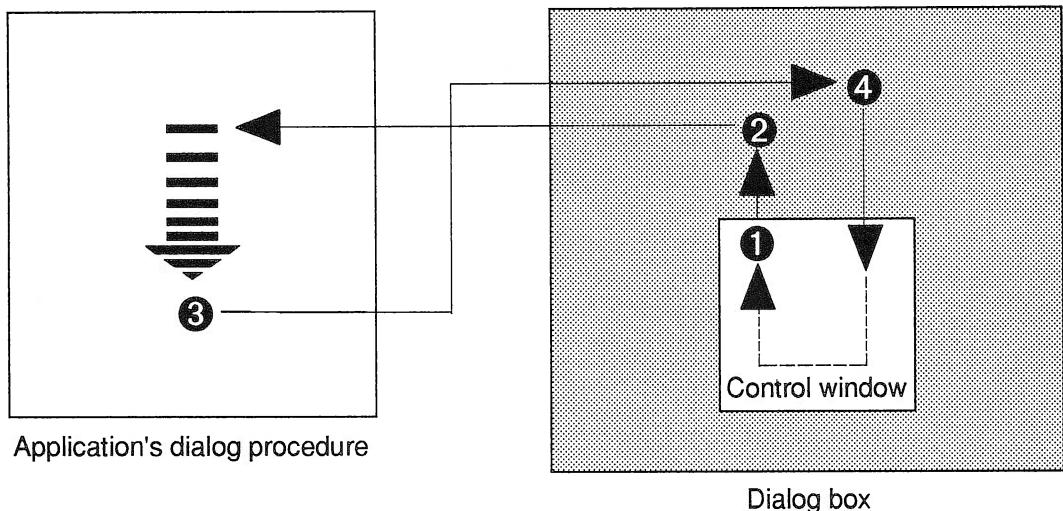
The Dialog Process Message Flow

Figure 10-6 shows the normal flow of messages and commands between the program's dialog procedure, the dialog box and a dialog control window. As shown in this Figure, some user interaction begins the process, for example selecting an item from a list box or selecting a push button.

- 1) The control window sends a message to the dialog box procedure indicating the nature of the user action.
- 2) The dialog box procedure sends a notification message to the program's dialog procedure indicating the nature of the user action.
- 3) The program's dialog procedure determines that some reaction is required, and sends on command to the control window via the dialog box procedure.
- 4) The dialog box procedure in turn sends a command to the control window to perform the action requested by the program.

This messaging process can also begin with the program's dialog procedure. The program does not need to wait for a dialog message. But, even when the process is reversed (with the order 3-4-1-2), the order of the flow is the same; only the starting point is altered.

Once started, this flow is continuous; the entire process is driven by the user's interaction with the dialog's control windows. As a result, it is not possible to designate a single starting and ending point as my simple example implies. The process of notification and command is continuous from the time the dialog is created until the dialog procedure issues the WinDismissDlg call to terminate the dialog process.



- ① The control window notifies the dialog box procedure of a significant event or the queried data is returned.
- ② The dialog box procedure sends a message to the application's dialog procedure notifying it of the event or returning the queried data.
- ③ The application's dialog procedure sends a control message to the dialog box procedure to alter or query the control window.
- ④ The dialog box procedure sends a message to the control window to alter its status or query its contents.

Figure 10-6 The dialog process message flow

Coding The Dialog Procedure

The structure of a dialog procedure is identical to the structure of a window procedure. You would expect this to be the case as dialogs are merely windows with special capabilities. The dialog procedure begins with the standard entry point defining the structure of messages passed to the procedure. Following the entry point is the COBOL Evaluate statement, beginning the process of intercepting messages of interest to the dialog procedure. As with window procedures, unwanted messages must be returned to PM using the default message call, WinDefDlgProc, the sole function under the Evaluate statement's *when other* phrase. The function of the WinDefDlgProc call is identical to the WinDefWindowProc call. As with window procedures, failure to return unwanted messages to the default dialog procedure will result in a loss of the normal dialog box functions furnished by the dialog box procedure.

Processing the flow of messages between the dialog box and the dialog procedure depends almost entirely on the individual dialog. For the sample program's Program Termination dialog, the processing is very simple. However, as more control windows are used and more information is collected from the user, dialogs can become very complex. In the next chapter, a more complex dialog, involving the processing and passing of user-entered data, will be added to the sample program.

For the program termination dialog, the sample program only needs to intercept the message generated when the user presses one of the two push buttons at the bottom of the dialog box. Push button notification is transmitted via the WM-COMMAND message and, as in prior chapters, is intercepted using the *when WM-COMMAND* phrase of the COBOL Evaluate statement. There is no need to check the WM-COMMAND message for the message ID as would normally be the case, as the program termination dialog box offers the user only these two push button control choices. This limited choice means the dialog procedure need only intercept this message, extract the push button resource ID (MsgParm1w1) and pass this ID back to the WM-QUIT routine. All other messages are passed back to PM using the WinDefDlgProc call.

It is important to note that the dialog's return code is passed in a special variable, DlgReturn. The normal Mresult variable cannot be used, as the continuous flow of messages to the dialog procedure, even after the dialog box is dismissed, can change the contents of Mresult before control is returned to the main program at the WinDlgBox call. As a result, the push button resource ID is stored in the variable DlgReturn before the WinDismissDlg call is issued.

Coding The WinDefDlgProc Call

The WinDefDlgProc call is coded exactly the same as the WinDefWindowProc call. Both calls perform the same function with only the target PM procedure different.

To code the WinDefDlgProc call, place the message structure defined in the entry point as the first four parameters, you are simply passing on the same message that was passed to the dialog procedure. Parameter one is the variable holding the handle of the dialog box, parameter two is the message ID and parameters three and four are the message's two parameters. The last parameter is the variable that will receive the return code from the default processing procedure. Here is the WinDefDlgProc call used in the sample program:

```
016620      Call OS2API 'WinDefDlgProc' using
016630          by value hwnd
016640          by value msg
016650          by value MsgParm1
016660          by value MsgParm2
016670          returning Mresult
```

Ending The Dialog

Only the user can end a dialog. This does not mean that a dialog must end only on user input; significant processing may occur between the user action and the end of the dialog. What it does mean is that only the user can indicate when a dialog is complete. Even when user input is in error and additional user interaction is required, the dialog must remain active until the user again indicates that the dialog is complete.

Every dialog has one control or set of controls that indicates to the user how to signal the end of the dialog. Only when one of these controls is selected can the dialog shutdown processing begin. Once the user signals the end of the dialog and the dialog procedure has validated the data entered, the dialog can be ended and the data passed back to the main program. For the Program Termination dialog, the end of dialog is signaled when the user selects one of the two push buttons. Regardless of its complexity, every dialog must have this easily identified ending control.

Dialog shutdown processing can be very complex, depending upon the amount of data entered by the user, but at the end of the processing when the procedure is finished communicating with the user, there must be a call to PM to hide or destroy the dialog box. WinDismissDlg destroys modal dialog boxes and hides modeless dialog boxes. In both cases, the user can no longer see or interact with the dialog, but in the case of a hidden dialog it can be invoked again without the overhead required to rebuild the dialog. Hidden dialogs are active windows and consume significant resources so dialogs should only be hidden when they will be reused in a relatively short period of time.

Coding The WinDismissDlg Call

The WinDismissDlg call requires only the handle of the dialog box and, as with the WinDefDlgProc call, this handle is available in every message sent to the dialog procedure. Simply code the variable declared in the procedure's entry statement as the message handle as parameter one. Parameter two is the variable containing the data

that will be returned by the dialog procedure to the WinDlgBox call. This does not mean that a dialog procedure can only return a single piece of data to the owning procedure; clearly complex dialogs must pass significant amount of data back and forth to their owning process. Only a single UShort value may be returned to the WinDlgBox call and the variable holding its value is placed in parameter two of the WinDismissDlg call.

Traditionally, this single value is the resource ID indicating how the user ended the dialog. From this value, the owning process should be able to determine if the dialog was completed successfully or not. This is not a hard and fast rule - any value can be returned by the dialog procedure - but user-entered data is usually placed into an application-defined data area, defined by the owning procedure and passed via the WinDlgBox call. For the program termination dialog there is only the resource ID of the user-selected push button to return, so it is returned in parameter two of the WinDismissDlg call.

Here is the code used in the sample program to end the program termination dialog:

```

016530      when WM-COMMAND
016540          Move MsgParm1w1 to DlgReturn
016550          Call OS2API 'WinDismissDlg' using
016560              by value hwnd
016570              returning DlgReturn

```

Interpreting The Dialog Results

With the dialog dismissed and the user data returned by the dialog procedure, the program must analyze the data to determine the next course of action. The ways to do this are as varied as the types of data that can be entered into a dialog box. It is not my intention to discuss how to analyze the data entered by the user. These functions are not unique to PM, and most COBOL programmers are well versed in these types of routines. I should emphasize, however, that analysis of the user's data, other than validating the input, should be restricted to after the dialog is complete. Response time during a dialog is extremely important. It should not take longer than .3 of a second to respond to user input. To stay within this time period, the dialog procedure must be kept as simple as possible.

For the sample program, the data analysis is limited to determining which push button the user selected. If the user selected the *OK* push button (*ID-TermOK*), indicating that the program should end, then the *program-done* flag is set and the message processing perform is ended. If the user did not select the *OK* button, then the *program-done* flag is not set and message processing continues. Here is how the sample program processes the data returned from the program termination dialog:

```
007940      If DlgReturn = ID-TermOK
007950          set program-done to true
007960      end-if
```


Chapter 11

More About Dialogs

Chapter 10 began the process of working with dialogs by introducing a simple dialog that, in response to the WM-QUIT message, asks the user to verify that the sample program may end. If the user selects the *OK* button, *program-done* is set to true and the program ends normally. If the user selects the *No* button, no action is taken and the program returns to the WinGetMsg call to wait for the next message. This simple dialog introduced the basic functions required with every dialog:

- Creating the dialog template.
- Building and destroying the dialog box.
- Coding a basic dialog procedure.
- Processing dialog messages.

The dialog procedure built in Chapter 10 includes the basics, but lacks support for the normal conversation that takes place between the user and dialog box on one hand, and the dialog box and dialog procedure on the other. With the Program Termination dialog, the user is allowed one choice, *OK* or *No*, and making that choice terminates the dialog. Most dialogs, however, involve a significantly greater interaction between the user and the dialog procedure. This ongoing interaction produces a stream of notification messages from the dialog box to the program's dialog procedure, and control messages from the program's dialog procedure to the dialog box. It is through this passing of the notification and control messages that a dialog procedure converses with the user.

This chapter introduces interactive dialogs by creating a dialog box containing a list box of PM calls and allowing the user to select one of the calls for display in the Source Code window.

The Types Of Dialog Control Windows

There are three types of dialog box control windows: frame windows, static control windows and interactive control windows. Each generates a different type of notification message.

Dialog frame windows are similar to standard frame windows. They communicate with the dialog routine using standard frame window messages - minimize, maximize, activate, paint, move, size, etc. The amount and type of frame messages a dialog procedure can receive are dependent upon the dialog itself. The more flexibility added to the dialog frame window, the more frame notification messages the dialog procedure will receive. If, for example, the System Menu is included as part of the dialog frame, then the dialog procedure will be subject to minimize, maximize, close, move and size notification messages.

Static control windows are fixed displays that do not allow user interaction and thus do not send notification messages to the dialog procedure. However, static control windows are still control windows and while the user cannot interact with them, the dialog procedure can, usually by displaying text or graphical characters in these windows. Text and group windows are examples of static control windows.

Interactive control windows are windows whose contents may be manipulated by the user. User interactions are through text entry, list, button, and box selection, scrolling, direct manipulation or any combination of these. With the exception of scroll bar windows, interactive control windows communicate with the dialog procedure using the WM-CONTROL message. The scroll bar windows use their own WM-HSCROLL and WM-VSCROLL messages. Since most user interaction with a dialog box is through interactive control windows, this chapter concentrates on the use of the interactive control window, and the resulting communication between the dialog box and the owning dialog procedure.

Interactive dialog procedures are a process of receiving notification messages and sending control messages in reply. But, the order in which notification messages are received by the dialog procedure cannot be guaranteed. Notification messages are generated as a result of user interaction with dialog control windows a user is free to move about the dialog as he sees fit. So, the dialog procedure must be structured to allow for the random arrival of notification messages. Editing user input that arrives in a non-sequential order can be especially difficult, but all dialog procedures must be prepared to deal with this random arrival of user input. See Figure 11-1 for a description of the dialog conversation required for the PM Call dialog.

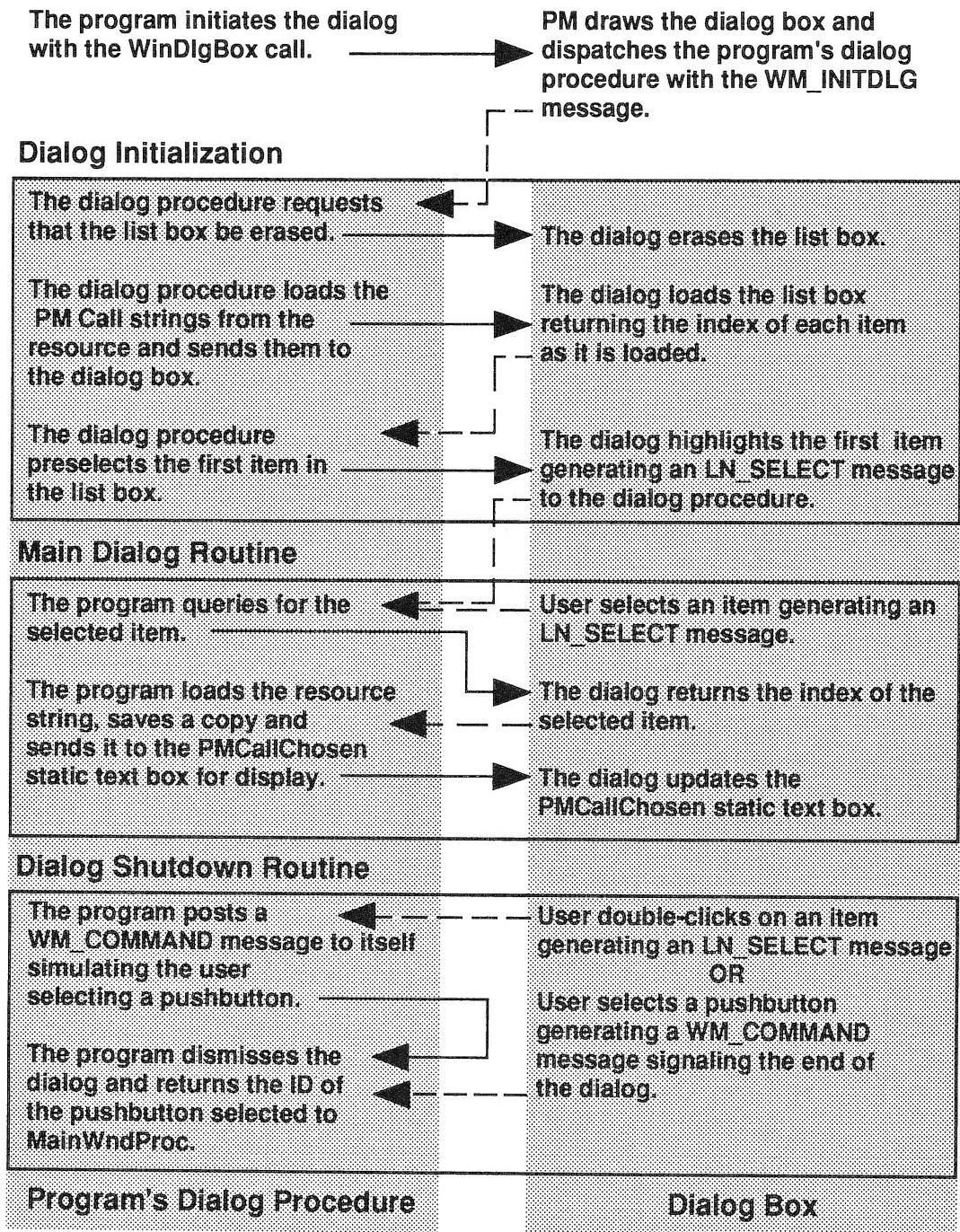


Figure 11-1 Communications between a dialog procedure and the dialog box

The Notification Messages

Control windows report significant events to the dialog procedure via the WM-CONTROL message. Parameter one of the WM-CONTROL message contains two short integer values containing the identity of the control window that generated the message and the message ID defining the exact message. Parameter two contains the handle of the control window generating the message. The following chart shows the unique notification message prefixes for each type of control window.

NOTIFICATION MESSAGE PREFIXES

Buttons	BN-
Checkboxes	
Push Buttons	
Radio Buttons	
User Buttons	
Combo Box	CBN-
Container	CN-
Entry Field	EN-
Frame	WM-
List Box	LN-
Menu	WM-
Multi-line Entry Field	MLN-
Notebook	BKN-
Scroll Bar	WM-
Slider	SLN-
Spin Button	SPBN-
Static	*
Title Bar	WM-
Value Set	VN-

* Static control windows do not generate notification messages.

Control Messages

The dialog procedure sends its commands and queries to the dialog control windows via dialog box control messages sent using the WinSendDlgItemMsg call. The dialog box, in turn, passes the message to the proper control window using the control window ID passed as part of the WinSendDlgItemMsg call. The contents of the message's two parameters varies by message. The following chart shows the unique control message prefixes for each type of control message.

CONTROL MESSAGE PREFIXES

Buttons	BM-
Checkboxes	
Push Buttons	
Radio Buttons	
User Buttons	
Combo Box	CBM-
	EM-
	LM-
Container	CM-
Entry Field	EM-
Frame	WM-
List Box	LM-
Menu	MM-
Multi-line Entry Field	MLM-
Notebook	BKM-
Scroll Bar	SBM-
Slider	SLM-
Spin Button	SPBM-
Static	SM-
Title Bar	TBM-
Value Set	VM-

Passing And Receiving Dialog Data

Dialog boxes are severely limited in the type and amount of data that can be passed between themselves and the program's dialog procedure by way of the WinDlgBox or WinLoadDlg calls. Without an additional data structure, no data can be passed to the dialog, and the dialog is limited to returning a single unsigned short integer as part of the WinDismissDlg call. Clearly, there is a need for larger amounts of data to be passed between the dialog and the program's dialog procedure. This chapter introduces a structured method for passing data between the dialog and its owing procedure.

When data needs to be passed to, or received from, a dialog box, the programmer must create a data structure with the format of CREATEPARAMS. See the *Presentation Manager Programming References* for more information on the CREATEPARAMS data structure. This SAA conforming application-defined data structure may be used to send initialization data to the dialog, receive data from the dialog or a combination of both. The data structure has a mandatory header composed of an unsigned short integer containing the length of the structure followed by an unsigned short integer containing

the type of structure. The structure length must be set the length of the data area plus 4. The data area of the structure may contain as many variables of whatever type and size as are required by the program. Here is the format of the CREATEPARAMS data structure:

```

01 Data Structure.
  05 StructLength      pic 9(4) comp-5.
  05 StructType        pic 9(4) comp-5.
  05 StructData1       pic x(xx).
  05 StructData2       pic 9(4) comp-5.
  .
  .
  .
  .
  05 StructDatax       pic x(xx).

```

Passing The Data Structure Pointer

The sample program's PMCall data structure is within the addressable area of the dialog procedures and may be addressed directly by the dialog procedures using simple COBOL move statements. But this is not usually the case with more complex programs, or dialogs that are contained within dynamic link libraries. For these dialogs, addressability to the data structure must be established each time the data structure is accessed by the dialog. PM supports this execution-time linking by allowing the program to pass a pointer to the CREATEPARAMS data structure as message parameter two of the WM-INITDLG dialog initialization message. The dialog must save this address pointer and establish addressability to the data structure.

For COBOL programs, addressability does not remain for the duration of the dialog, but is broken each time the dialog procedure is exited. Addressability must be reestablished every time the procedure is entered. The need to reestablish addressability requires that the dialog procedure save the pointer in an area of memory to which it has permanent addressability. Local-Storage may not be used to save the data structure pointer. Dialog procedures with their own Working-Storage can simply save this pointer into a pointer variable within Working-Storage. For dialogs without their own Working-Storage, such as the PM Call dialog, or for dialogs that manage multiple windows, a solution is not so simple.

To aid in storing of this type of window related data, PM provides a special area of window storage called window words. Window words are an area of PM managed storage set aside for each window. Most of this storage is used by PM, but some window words are available for use by the program. A window words is the correct place for storing window

related data such as the data structure pointer. But, window words will not be discussed in detail until Chapter 15, so for the present I will simply store the pointer to the CREATEPARAMS structure back into the Main program's Working-Storage. Remember however, that window words is the correct place to store this pointer.

Even if the data structure is not accessed during initialization processing, the WM-INITDLG message processing routine must save the data structure pointer. The pointer is passed only within this initialization message. The COBOL Set statement is used to save the pointer into Working-Storage. Here is the Set statement used in the sample program to save the pointer in the Working Program variable *PMCallPointer*:

```
016950      Set PMCallPointer to Msg2Pointer.
```

The dialog's entry statement defines message parameter two as an unsigned long integer supporting most PM messages. To receive the CREATEPARAMS data structure pointer, message parameter two must be redefined within the Linkage Section as a pointer. Here is the redefinition of message parameter two for use with the WM-INITDLG message:

```
004600 01 MsgParm2          pic 9(9) comp-5.
004610 01 Redefines MsgParm2.
004620 05 MsgParm2w1        pic 9(4) comp-5.
004630 05 MsgParm2w2        pic 9(4) comp-5.
004640 01 Msg2Pointerusage is pointer redefines MsgParm1.
```

Establishing Structure Addressability

Addressability is established by first defining a data structure within the dialog's Linkage Section that is identical to the Working-Storage data structure. Then, using the COBOL Set Address statement, the Linkage Section data structure is assigned to the same memory location as the Working-Storage data structure. Once addressability is established, the dialog procedure places data into the Linkage Section data structure, automatically making it available to the main procedure at the Working-Storage data structure. The reverse is also true. Data placed into the Working-Storage data structure is available to the dialog procedure at the Linkage Section data structure once addressability is established. Here is the Working-Storage data structure followed by the Linkage Section data structure used for the PM Call dialog:

```

002890 01 PMCall.
002900      05 PMCLength          pic 9(4) comp-5 value 39.
002910      05 PMCType           pic 9(4) comp-5 value 0.
002920      05 PMCReturn          pic 9(4) comp-5 value 0.
002930      05 PMCL listBoxSize   pic s9(4) comp-5 value 0.
002940      05 PMCL listBoxEntry  pic x(31).

004710 01 DPMCall.
004720      05 DPMCLength         pic 9(4) comp-5.
004730      05 DPMCType          pic 9(4).
004740      05 DPMCRetn           pic 9(4) comp-5.
004750      05 DPMCL listBoxSize  pic s9(4) comp-5.
004760      05 DPMCL listBoxEntry pic x(31).

```

Every PM message processing routine that accesses the data structure must first establish addressability by issuing the COBOL Set Address statement. This statement uses the *PMCallPointer* pointer saved during the dialog's initialization processing to correctly position the Linkage Section data structure. See Figure 11-2. Here is the Set Address statement used to establish data structure addressability:

```
016960           Set Address of DPMCall to PMCallPointer.
```

The PM Call Dialog

The PM Call dialog added in this chapter presents a list box containing the PM calls utilized in the sample program and allows the user to select one of the calls for display within the Source Code window. There is no restriction on the number of times the user can select an entry from the list box, but the last call selected when the dialog completes is used for the display. When a call is selected from the list box, it is displayed within a static group box in the upper left-hand corner of the dialog box identifying it as the currently selected call. The user ends the dialog in one of two ways. Selecting one of the push buttons ends the dialog, either *OK* to use the last call selected for display or *Cancel* to end the Source Code display. Double-clicking on any entry in the list box selects that entry for display and ends the dialog as if the *OK* push button had been selected.

For each selection the user makes, the program's dialog procedure receives a WM-CONTROL message with the LN-SELECT message ID. The dialog procedure then queries the list box for the index of the selected item using the WinSendDlgItemMsg call with the LM-QUERYSELECTION ID. The list box returns the zero-based index number

of the selected item as the call's returned data. The dialog procedure uses this index to locate and load the correct text string from the program's resources with the WinLoadString call, then displays the string in the PMCallChosen text window using the WinSetDlgItemText call (see Figures 11-3 and 11-4).

Creating The PM Call Dialog Template

You should be familiar with the Dialog Editor and creating dialog templates after completing Chapter 10. I will not spend time here detailing how to create the PM Call dialog box template. If you feel unsure about creating a dialog box template, please review Chapter 10 as well as the *Presentation Manager Programming References*. See Figure 11-5 for a list of the controls used in the PM Call dialog box. The complete PM Call dialog box template together with the related resource definitions are included in Appendix A.

Updating The Resource Script File

As with any dialog, the Resource Script file must be updated to include a reference to the new PMCall.Dlg source template and PMCall.h header file.

If the DLGINCLUDE statement has been removed from the dialog template, a `#include` reference to the dialog's header file must be added to the Resource Script file. A RCININCLUDE statement specifying the PMCall.Dlg dialog template must be inserted at the end of the Resource Script file to allow the Resource compiler to include the PMCall dialog template in the Resource file.

In addition to the actual dialog template, the text strings that will be displayed in the PM Call dialog's list box must be defined. These strings are stored in the Resource file within a string table and their resource values defined in the Sample.h Resource Header file. These strings could have been added to the existing string table, but I chose to use a second string table for clarity. Here are a few table entries from the sample program's Resource Script file. The complete Resource Script file is included in Appendix A.

```

STRINGTABLE
BEGIN
  IDS_PMCALL1,    "GpiCharStringAt"
  IDS_PMCALL2,    "GpiCreateLogFont"
  IDS_PMCALL3,    "GpiDeleteSetId"
  IDS_PMCALL4,    "GpiErase"
  IDS_PMCALL5,    "GpiQueryFontFileDescription"
  .
  .
  .
  .
IDS_PMCALL42,    "WinWindowFromID"
END

```

Updating The Program's Working-Storage Section

As with other additions to the Resource Script file, duplicate declarations must be made in the program's Working-Storage Section for those resources that are referenced within the program. Only the first of the PM call strings needs to be declared, as the value of any individual string can be calculated based on the index value of the call selected by the user. Additionally, the list box notification and control message IDs and the data passing structure *PMCall* must be declared in Working-Storage. The mirror image of this data structure, *DPMCall*, must be declared in the Linkage Section. Slightly

See Figure 11-2 at Right

- ① **MsgParm2 is redefined as the pointer Msg2Pointer.**
- ② **The WinDlgBox call passes a pointer to the PMCall data structure to the dialog procedure.**
- ③ **The WM_INITDLG message carries the pointer to the PMCall data structure as message parameter 2.**
- ④ **The Set statement loads the pointer and positions the DPMCall data structure labels to the same memory locations as the PMCall data structure.**
- ⑤ **The structure DPMCall and PMCall now occupy the same memory locations. Data placed into DPMCall is available to the Main program at PMCall.**

Main Program

```

Working-Storage Section.
01 PMCall.
  05 PMCLength          pic 9(4) comp-5 value 39.
  05 PMCType            pic 9(4) comp-5 value 0.
  05 PMCReturn          pic 9(4) comp-5 value 0.
  05 PMCLListBoxSize   pic s9(4) comp-5 value 0.
  05 PMCLListBoxEntry  pic x(31).
  :
  :
Call OS2API 'WinDlgBox' using
  by value      HWND-DESKTOP
  by value      hwndClient
  by value      WindowProc
  by value      ShortNull
  by value      ID-PMCallDlgBox
  by reference  PMCall ②
  returning    DlgReturn
  :
  :

```

Dialog Procedure

```

Linkage Section.
01 Msg2Pointer usage is pointer redefines MsgParm2. ①

01 DPMCall.
  05 DPMCLength         pic 9(4) comp-5.
  ⑤ 05 DPMCType          pic 9(4) comp-5.
  05 DPMCReturn          pic 9(4) comp-5.
  05 DPMCLListBoxSize   pic s9(4) comp-5.
  05 DPMCLListBoxEntry  pic x(31).
  :
  :

entry 'PMCallDlgProc' using
  by value hwnd
  by value msg
  by value MsgParm1
  by value MsgParm2 ③

Set address of DPMCall to Msg2Pointer ④

```

Figure 11-2 Linking the dialog data passing structure

different structure and variable names for the two data areas are required only when they both appear within the same compile. Normally, the matching data structures and variables have the same names. All of these resources are shown in the sample program included in Appendix A.

Building The PM Call Dialog Box

The PM Call dialog box is displayed when the user selects *Source* from the *View* pull-down menu. To create and display the PM Call dialog box, the *WinDlgBox* call must be inserted as the first function performed when the *MI-Source* command message is received.

The *WinDlgBox* call loads, builds and displays the PM Call dialog box as an application modal dialog. This call then suspends the *MI-Source* processing routine and transfers control to the PM Call dialog procedure at the entry point address passed as parameter three of the *WinDlgBox* call. The *MI-Source* processing routine remains suspended until the PM Call dialog procedure issues the *WinDismissDlg* call returning control to the Main window procedure.

To code the *WinDlgBox* call, specify the handle of the parent window as parameter one and the handle of the owner window as parameter two. As with the Program Termination dialog, make the parent of the dialog the desktop. There is some flexibility when specifying the dialog's owner, by as the Main window's Client window is the active window when the dialog is created, I made it the owning process. Parameter three is the entry point address of the dialog routine. This address must be loaded using the COBOL Set statement prior to the execution of the *WinDlgBox* call. Parameter four is the name of the module holding the dialog template, if the template is contained within a dynamic link library. As the sample program's resource is bound with the program, parameter four is coded as a null. Parameter five is the dialog template identity.

Parameter six is new with this iteration of the *WinDlgBox* call. This parameter is the address of the dialog data structure used to send and receive data from the dialog procedure. Coding the *by reference* phrase together with the 01 level of the data structure causes the address of the data structure to be passed to PM and then on to the dialog procedure as part of the WM-INITDLG message. The last parameter is the variable that is to receive the returned data from the dialog routine. As with the Program Termination dialog, a special field must be used for this return value. The standard *Mresult* variable may not be used to return this value. See Chapter 10 for a more detailed description of the *WinDlgBox* call.

Here is the *WinDlgBox* call to create the PM Call dialog box:

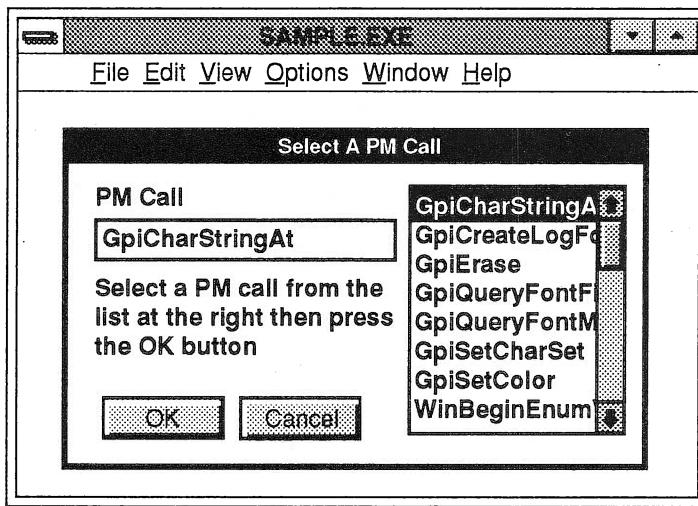


Figure 11-3 The Main Window with the Select PM Call dialog box (Version 1.3)

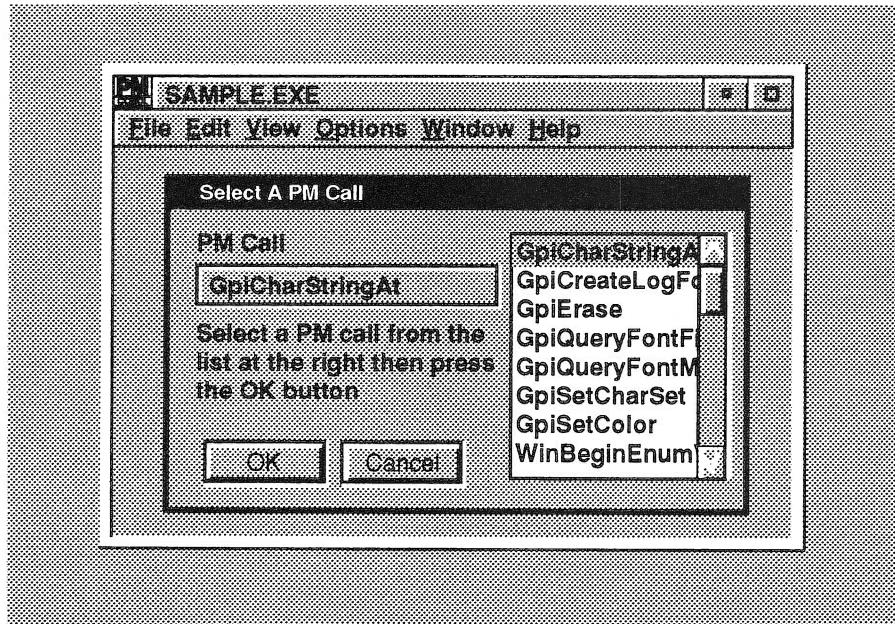


Figure 11-4 The Main Window with the Select PM Call dialog box (Version 2.0)

```

010840      set WindowProc to Entry 'PMCallDlgProc'
010850      Call OS2API 'WinDlgBox' using
010860          by value      HWND-DESKTOP
010870          by value      hwndClient
010880          by value      WindowProc
010890          by value      ShortNull
010900          by value      ID-PMCallDlgBox
010910          by reference PMCall
010920          returning     PMCReturn

```

The Dialog Procedure Initialization Routine

Before a dialog box is displayed to the user, PM passes control to the dialog procedure via the WM-INITDLG message to let the dialog perform any initialization processing that may be required. Initialization processing includes functions such as filling of list boxes and multi-line entry boxes, preselection of buttons and check boxes, preloading and establishing edit control over entry fields, positioning of the dialog box on the desktop, and establishing addressability to the data passing structure, if used.

The WM-INITDLG message processing routine in the PM Call dialog procedure performs the following functions:

- Establishes addressability to the PMCall data passing structure.
- Saves the data structure pointer.
- Loads the PM call text strings into the list box.
- Preselects the first item in the list box.
- Positions the dialog box in the center of the desktop.

After saving the pointer and establishing addressability to the data structure, the initialization routine loads the PM call text strings into the list box. However, before the list box can be loaded, any entries that may remain from a prior dialog must be erased from the list box array. List box entries are held in an array managed by PM and are a PM resource. So, to be sure the list box array is empty, it is good procedure to always erase the array just prior to every loading.

The list box array is erased by sending the LM-DELETEALL command to the list box control window using the WinSendDlgItemMsg call. This call is coded by placing the handle of the dialog in parameter one. As with other dialogs, this is the handle of the current message. Parameter two is the resource identity of the target control window. For the sample program, this is the ID of the list box, *ID-PMCallListBox*. Parameter

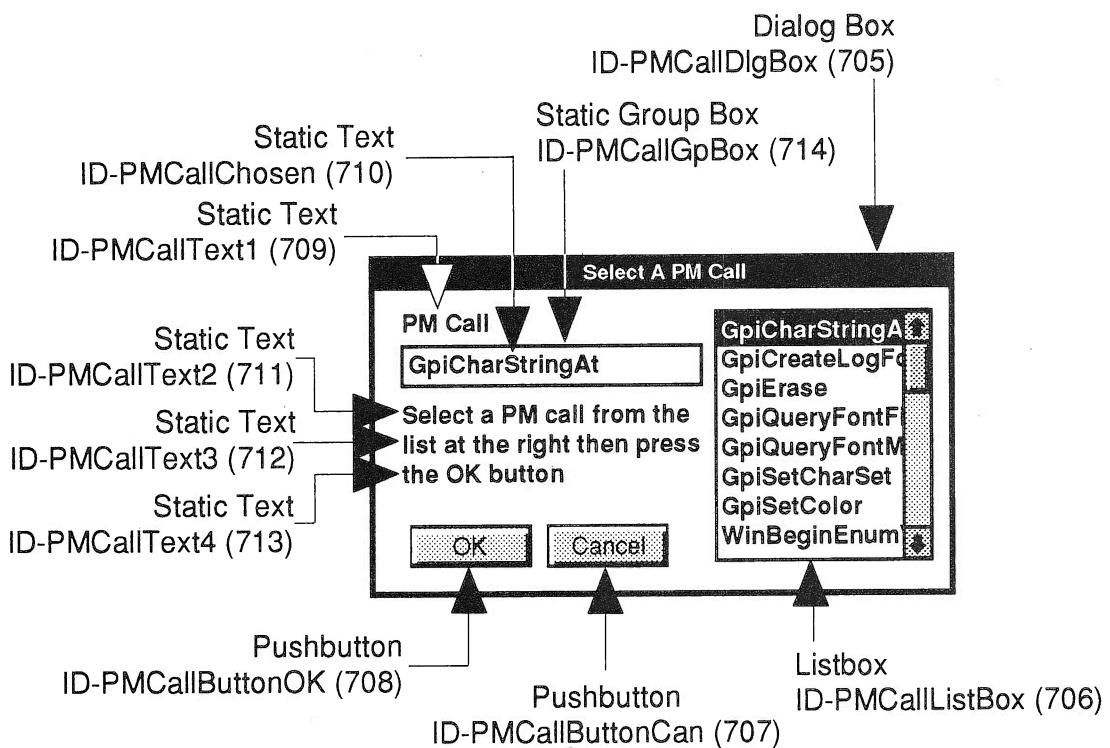


Figure 11-5 Resources for the PM Call dialog box

three is the command to be executed by the list box, LM-DELETEALL. Parameters four and five are the two message parameters. For the LM-DELETEALL message these parameters are not used so they are coded as null. The last parameter is the variable that will receive the call's return code. Here is the WinSendDlgItemMsg call to send the LM-DELETEALL message to the list box:

```

017000          Call OS2API 'WinSendDlgItemMsg' using
017010            by value hwnd
017020            by value ID-PMCallListBox
017030            by value LM-DELETEALL
017040            by value LongNull
017050            by value LongNull
017060          returning ReturnData

```

Here is the structure of the LM-DELETEALL message:

THE LM-DELETEALL MESSAGE				
Message Id	MsgParm1w1	MsgParm1w2	MsgParm2w1	MsgParm2w2
(Hex) 016E	0000	0000	0000	0000
(Dec) 0366	0000	0000	0000	0000
LM-DELETEALL				
Null				
Null				

Loading The List Box

With a guaranteed empty list box array the PM call strings can be loaded. The PM call strings are first retrieved from the program's resources, then sent to the list box, one at a time. A COBOL perform handles this function in the sample program. This routine does not contain any new calls, but it does illustrate an additional list box command.

As each string item is sent to the list box for loading, the list box must be told how to load it. There are three ways an item may be added to a list box. The item may be inserted at the end of the existing entries (LIT-END), it may be inserted in ascending order within the existing entries (LIT-SORTASCENDING) or it may be inserted in descending order within the existing entries (LIT-SORTDESCENDING). Loading new entries at the end of the existing entries (LIT-END) is the fastest way to load a list box, but it places a burden upon the program to keep the items in proper sequence. Regardless of how entries are added to a list box, they must be kept in an order meaningful to the user. Simply looking at the displayed list box items should indicate to the user the scroll direction of the next required item. For the sample program, each item is inserted into the list box at the end of the existing strings (LIT-END). The order in which they are defined in the Resource insures they will be loaded in ascending order.

The text strings are transferred to the list box control window by passing a pointer to the string in the LM-INSERTITEM dialog control message. LM-INSERTITEM, as are other dialog messages, is sent using the WinSendDlgItemMsg call. The WinSendDlgItemMsg call is coded exactly the same as the prior use of the call with these three differences. Parameter three contains the command directive for this call, LM-INSERTITEM. Parameter four (message parameter one) indicates the item should be inserted after the last item loaded and parameter five (message parameter two) contains the pointer to the text string. As each item is loaded, the list box returns the zero-based index of the item within the list box array.

Here is the WinSendDlgItemMsg call used to load the list box in the PM Call dialog. Note that this call is conditioned on the loading of a valid string from the resources. If the

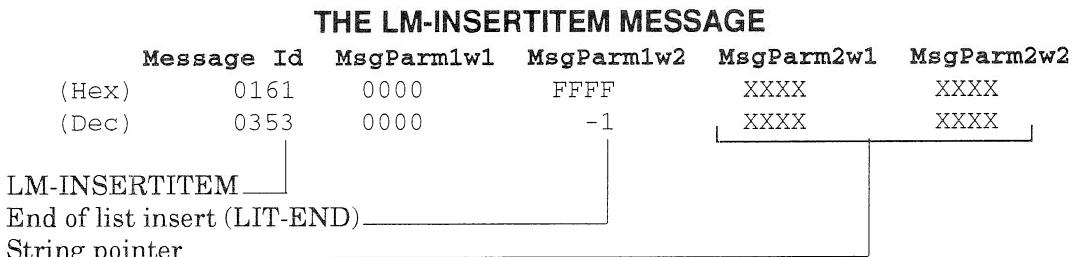
WinLoadString call returns 0 as the length of the string, then the WinSendDlgItemMsg call is not executed and the perform ends.

```

018390      If DPMCL listBoxSize > 0
018400          Call OS2API 'WinSendDlgItemMsg' using
018410              by value      hwnd
018420              by value      ID-PMCallListBox
018430              by value      LM-INSERTITEM
018440              by value      LIT-END
018450              by reference  ListBoxEntry
018460                  returning   ReturnData
018470      end-if.

```

Here is the format of the LM-INSERTITEM message:



After loading the list box, the initialization routine preselects an item in the list. The item preselected should be the item most likely to be chosen by the user, but in fact, any item may be selected using any criteria. The sample program simply selects the first item in the list box. A list box item is selected by sending the LM-SELECTITEM message to the list box control window. As with other control messages, the WinSendDlgItemMsg call is used to send the message. For item selection, the WinSendDlgItemMsg call requires the LM-SELECTITEM command as parameter three. Parameter four (message parameter one) contains the zero-based index number of the item to be selected. Since the sample program is selecting the entry at the top of the list box, a 0 is sent as the index number. Parameter five (message parameter two) is coded as *true* to indicate that the item should be selected. Coding parameter five as *false* would deselect the item. Here is the WinSendDlgItemMsg call used to preselect the first item in the list:

```

017170      Call OS2API 'WinSendDlgItemMsg' using
017180          by value hwnd
017190          by value ID-PMCallListBox
017200          by value LM-SELECTITEM
017210          by value 0 size 4
017220          by value 1 size 4
017230      returning ReturnData

```

Here is the format of the LM-SELECTITEM message:

THE LM-SELECTITEM MESSAGE					
Message Id	MsgParm1w1	MsgParm1w2	MsgParm2w1	MsgParm2w2	
(Hex) 0164	0000	0000	0000	0001	
(Dec) 0356	0000	0000	0000	0001	

LM-SELECTITEM _____

Index of item to be selected _____

Select item (True) _____

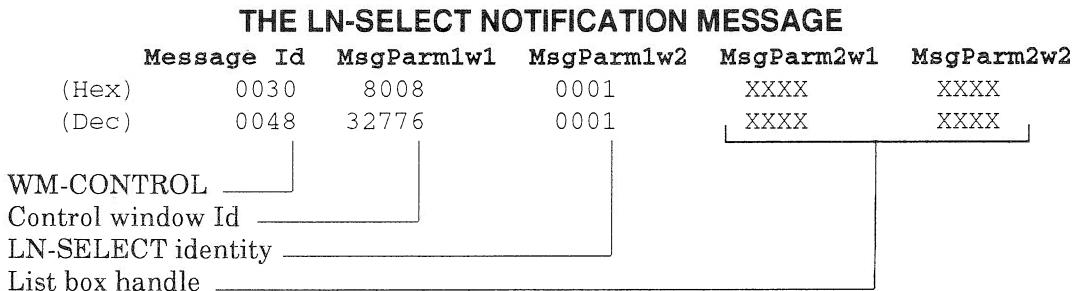
The preselected item must also be displayed as the currently selected item in the *ID-PMCallChosen* static text box. However, no logic is required in the initialization routine to display the item. The LM-SELECTITEM message generates a recursive LN-SELECT message, indicating that an item has been selected, that flows back through the dialog's main processing routine, where the selected string is moved to the *ID-PMCallChosen* control by the routine that handles normal user selections.

Processing The Dialog

Exiting the dialog procedure after processing the WM-INITDLG message causes PM to display the dialog box. From this point forward, the user is free to interact with the dialog's control windows, in any order, producing notification messages that are sent to the dialog procedure.

For the PM Call dialog, the user can interact with the list box producing two possible WM-CONTROL messages (LN-SELECT or LN-ENTER), or with the push buttons producing a WM-COMMAND message containing the resource ID of the button selected.

An LN-SELECT message, indicating that an item has been selected, is generated when the user single-clicks on one of the list box entries. If the LN-SELECT message is received, the dialog routine must send the LM-QUERYSELECTION message to the control window to determine which entry was selected. As before, this message is sent using the WinSendDlgItemMsg call. Here is the format of the LN-SELECT message:



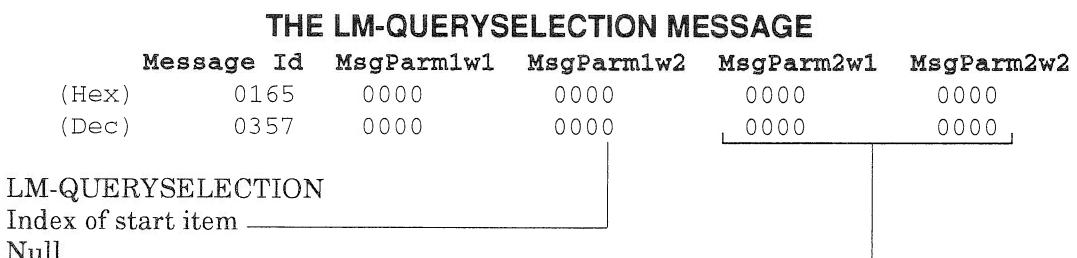
The LM-QUERYSELECTION returns the zero-based offset of the item selected. Adding this offset to the *IDS-PMCall1* resource value yields the resource value of the selected string. Here is the WinSendDlgItemMsg call used to query the index of the selected item:

```

017670      Call OS2API 'WinSendDlgItemMsg' using
017680          by value hwnd
017690          by value ID-PMCallListBox
017700          by value LM-QUERYSELECTION
017710          by value LongNull
017720          by value LongNull
017730          returning ReturnData

```

Here is the format of the LM-QUERYSELECTION message:



The correct string is then loaded from the resource into the data passing structure, and then displayed in the *PMCallChosen* text box using the WinSetDlgItemText call. By loading the string directly into the data passing structure, the last PM call selected by the user is available to the main procedure at the end of the dialog without additional processing.

The WinSetDlgItemText call is coded by placing the handle of the dialog in parameter one. As with other calls that require the dialog's handle, use the current message handle. Parameter two contains the resource ID of the field that is to receive the text string. For the sample program, this is the static text box *ID-PMCallChosen*. Parameter three is a pointer to the null-terminated text string to be displayed on the dialog. The last parameter is the variable that is to receive the call's return code. Here is the WinSetDlgItemText call used in the sample program to display the PM call chosen by the user:

```

017900      Call OS2API 'WinSetDlgItemText' using
017910          by value      hwnd
017920          by value      ID-PMCallChosen
017930          by reference DPMCLListBoxEntry
017940          returning    ReturnData

```

An LN-ENTER message is generated when the user double-clicks on one of the list box entries, indicating that an item has been selected and the dialog should be ended. For every LN-ENTER message generated, a corresponding LN-SELECT message is also generated. With the LN-SELECT message procedure retrieving and saving the selected text string, the LN-ENTER message procedure only needs to perform the dialog's end processing. However, as with the WM-QUIT processing, the end-of-dialog code already exists within the push button processing routine. So, rather than coding a second dialog end routine, the LN-ENTER procedure simply posts a WM-COMMAND message, with the ID of *PMCallButtonOK*, to itself to simulate the user pressing the *OK* push button. This message immediately transfers control to the WM-COMMAND routine within the dialog procedure, where the message is processed by the push button routine.

Ending The Dialog

As with the Program Termination dialog, ending this dialog involves loading the resource ID of the button selected by the user or generated by the LN-ENTER routine into the return variable *DPMCRetn* and issuing the WinDismissDlg call.

The WinDismissDlg call returns control to the main procedure at the WinDlgBox call with the selected button available for testing by the first instruction after the call. In addition, the *PMCall data structure* contains the PM call text string retrieved from the resource file by the dialog.

A note about the dialog return variable *DPMCReturn* is in order. With the Program Termination dialog I discussed the need for a dialog return variable separate from the normal procedure return variable. That requirement still exists, but the implementation changes with the introduction of a data passing structure. Declaring a single, shared variable in Working-Storage to be used as a common dialog return is poor practice for reentrant programs. As a result, the dialog return variable must be contained in the data passing structure along with the other unique dialog variables. This ensures a unique variable for each dialog.

The main procedure uses the PM call text string as the title of the Source Code window that will eventually display the selected PM call. To add the text to the child window, the main procedure issues the WinSetWindowText call immediately after the child window is created.

The WinSetWindowText is coded with the handle of the window frame to receive the title as parameter one. Parameter two is a pointer to the *PMCLListBoxEntry* area containing the null-terminated text title string. The last parameter is the standard return code variable. Here is the WinSetWindowText call used to add the title to the source listing child window in the sample program:

```
011150      Call OS2API 'WinSetWindowText' using
011160          by value      hwndSource
011170          by reference PMCallData
011180          returning    ReturnData
```


Chapter 12

Working With Text And Fonts

Programs running in a PM window have great flexibility in choosing the style of their text characters. Through the OS/2 Graphical Programming Interface and an appropriate font, applications have full control over the size, stroke, weight and shear of each character drawn into the presentation space. There are thousands of different type faces, each with variations in the way characters are formed, that when combined with PM's font manipulation techniques offers endless ways to display a program's text to the user.

This chapter introduces the concept of fonts, and shows how to use fonts to control the style of the text that PM displays to the user. This chapter is by no means a complete discussion of fonts. There are numerous books available that discuss the use of fonts in much greater detail than this single chapter, and you are encouraged to read at least some of this additional material.

A Word About Fonts

To help understand how OS/2 makes use of fonts, it is necessary to understand the difference between code pages, font, or type, faces and fonts. The distinction between these terms becomes important when working with the font metrics and font attributes of graphical text.

A code page is a translation table used to convert each of the 256 ASCII values into a specific letter, number or symbol. A code page translation is required for data routed to any display, printing or plotting output device. The code page does not define the look of each character, only the specific value of each ASCII byte. For example, code page 437 will translate the ASCII hex 68 into a lower case letter h, but the code page has no effect

on the style of the letter, only its value. The two primary code pages used in the United States are code page 437 and code page 850. There are others, of course, each with unique differences in their translations. The choice of which code page to use depends on the desired output.

After the code page has translated the ASCII bytes into characters, numbers or symbols, the font face selected for this output, controls the exact style of each character. It is the font face that makes the characters look the way they do. The New Century Schoolbook font face, for example, gives the text in this book the familiar look of a newspaper or magazine.

All type faces are divided into two categories, serif faces - meaning literally "little feet," but for font faces it defines a style with fine lines projecting from the main stroke of the letter - and sans serif (without serifs). The font used for this book is a serif font. Within each of these categories, the individual slant, weight and shear of the characters gives each face its distinctive look. Here are four lines of text. All four lines use the same code page, so each line contains the same characters. However, the characters in each line look different because each uses a different font face.

The is a sample line of text using code page 437.

The is a sample line of text using code page 437.

The is a sample line of text using code page 437.

The is a sample line of text using code page 437.

A font is an individual rendering of a font face. A font combines the exact size of the characters with the stroke, weight and shear defined by the font face. Therefore, it is incorrect to say *a Times Roman font*. What is correct is *a 10 point bold italic Times Roman font*, specifying the exact physical properties of the Times Roman face. Here are four lines of text. All four lines use the same code page, so each line contains the same characters. Each line uses the ITC Avant Garde Gothic font face, but each line is a different font because the size, shear, and weight of each line is different.

The is a line of text using the ITC Avant Garde Gothic font face.

The is a line of text using the ITC Avant Garde Gothic font face.

The is a line of text using the ITC Avant Garde Gothic font face.

The is a line of text using the ITC Avant Garde Gothic font face.

A font's size is normally measured in printers' points. Each point is equal to 1/72 of an inch, or 72 points to the inch. Thus, a 24-point font is 1/3 of an inch high (individual characters vary in height). The Presentation Manager, however, measures fonts in a much smaller increment, called twips. A twip is equal to 1/20 of a printer's point, or 1440

twips to the inch. To PM, a 24-point font measures 480 twips high. A program can change this internal measurement from twips to Pels, Lometric (0.1mm), Himetric (0.01mm), Loenglish (0.01 inch) or Hienglish (0.001 inch).

Fonts supplied as part of the OS/2 Operating System or added to the OS/2 system are classified as public fonts and are loaded when the system is started. Public fonts are available to any application running in the system. Conversely, a private font is defined by an individual program and is available only to the program that defines it. A private font must be loaded by the program using the GpiLoadFont call, before it can be used and unloaded using the GpiUnloadFont call when it is no longer needed. Because private fonts are available only to the program that defines them, private fonts cannot be used to print documents created using the standard data stream format (PM_Q_STD), as private fonts are not available to the print driver that builds the actual printer output. Private fonts may be used in documents printed using the raw data stream format (PM_Q_RAW). See Chapter 17 for a complete description of standard and raw data stream formats.

Anti-Aliased fonts are specialized fonts built for use with the PS/2 XGA display adapters and the XGA device drivers. Information about Anti-Aliased fonts can be found on the diskette that contains the device drivers. Anti-Aliased fonts will not be discussed in this book.

Image vs. Outline Fonts

Two types of fonts are available within the OS/2 graphical subsystem, image fonts (also called raster fonts) and outline fonts (also called vector fonts). The structure of these fonts has nothing to do with the font's face or style. Most of today's popular font faces are available in both image and outline form. But, the structure of the font determines how the characters are created within the presentation space. Each method has its strengths and weaknesses and an understanding of each method is important.

Regardless of the font used, image or outline, the characters generated are displayed or printed as a series of black or colored dots, called pels. So, the difference between image and outline fonts is not how their characters are displayed or printed, but the process of setting the correct pels for each character. Before any font can be used to display or print characters, it must be rendered. Rendering is the process of converting each character in the font's standard format into a series of dots that represents the characters in the size, weight and slant requested by the program.

Each character in an image font is stored as a series of dots referred to as a bit map. Within each bit map, selective dots are turned on to produce the image of the character in exactly the form that will be displayed. Each image font contains the correct

The Presentation Manager Image Fonts*			
Point Size	Roman	Swiss	Courier
8-point	✓	✓	✓
10-point	✓	✓	✓
12-point	✓	✓	✓
14-point	✓	✓	
18-point	✓	✓	
24-point	✓	✓	

* An additional proportional Swiss font is supplied for use in window components such as title bars and menus.

Figure 12-1 Available Presentation Manager image fonts

combination of dots to produce each character's image in a specific size, weight and slant. Because image fonts are stored as specific bit maps, they cannot be scaled, rotated, condensed or expanded. There must be a specific image font for each size, weight and slant to be used. There is, for example, a specific image font to generate a 10 point bold italic Times Roman font. While image fonts are inflexible, they are faster to display and print as very little rendering work is required, and they produce characters with a slightly better screen appearance at very small point sizes. See Figure 12-2.

Each character in an outline font is stored as a series of lines, angles and arcs that mathematically define the outline, or shape, of the character. Being composed of lines, angles and arcs, rather than dots, these characters can be scaled, rotated or sheared by simply altering the formulas that define them. Outline characters are rendered by first executing the formulas to create the desired outline of the character, then filling in the outline with the dots that will form the character. But, before the outlines are filled, the graphics text engine applies subtle changes, called hints, to the rendered characters, adjusting each character to account for errors introduced by the scaling and weighting of the characters. The hinting of outline characters gives a smoother, more recognizable appearance to the final characters than with an image font.

While outline fonts are somewhat slower, requiring significantly more work to render, and may appear more ragged at very small point sizes, their flexibility and smooth appearance at normal point sizes makes them the drawing method of choice for use with OS/2. However, the flexibility of outline fonts requires more complex program logic to control and manipulate the characters. As a result, I have chosen to introduce the concepts of fonts and the Graphical Programming Interface calls that support them using the simpler image fonts.

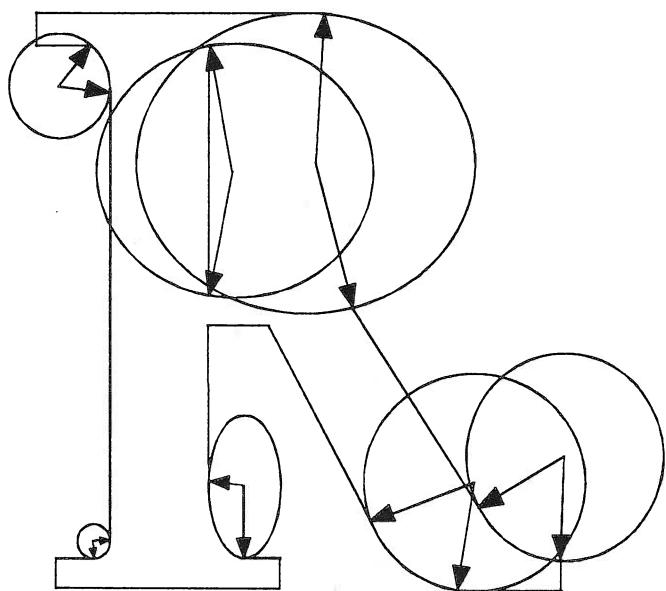
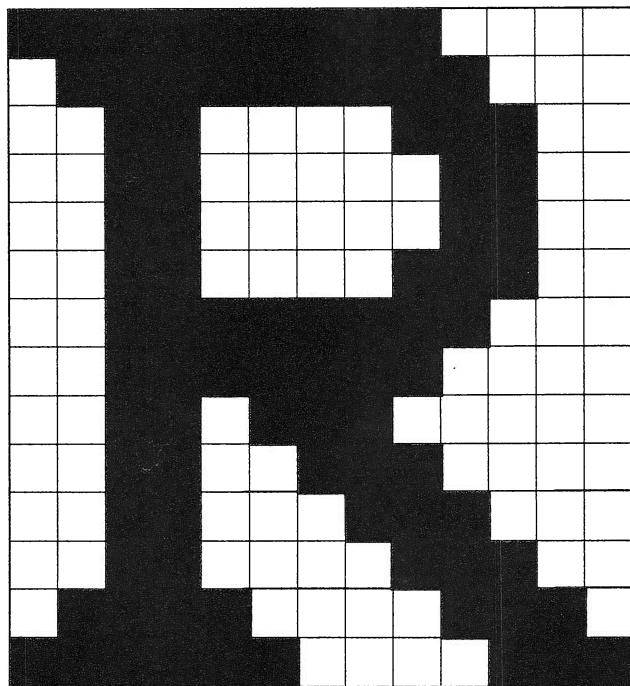


Figure 12-2 Comparison of an image font (top) and an outline font (bottom)

Selecting The Desired Font

Technically, a program cannot select the actual font or fonts that are to be used. Fonts are chosen only by PM based upon a list of desired attributes passed to PM in a Font Attribute Table. PM selects the font from the available public and private fonts that most closely match the application's requirements. This best fit selection allows programs to obtain the font that most closely matches their requirements, regardless of the fonts available at execution time. While PM does the actual font selection, it is not random and can be influenced by specifying sufficient parameters within the Font Attribute Table to narrow PM's choices to only those fonts the program wants. Specifying a font face, for example, will restrict PM to selecting fonts only with the specified face.

To ensure that the best possible font is chosen by PM, the following procedure should be used. Performing these steps will ensure that the program has true font independence, utilizing the best of the available fonts regardless of the execution environment.

- 1) Query for all available fonts within the desired type face using the GpiQueryFonts call with the required face name and the font count set to 0. The return code will contain the number of fonts available for the specified font face.
- 2) Allocate a Font Metrics Table large enough to contain the number of fonts returned by the query - occurs equal to the number of fonts returned. If dynamic memory allocation is not possible, then the preallocated table should be large enough to hold all the metrics of the specified font.
- 3) Reissue the GpiQueryFonts call with the font count set to the number returned by the first call.
- 4) Examine each font metric returned until the closest fit to the program's requirements is found.
- 5) Move the *Fmetrs-Match* and *Fmetrs-FaceName* values from the selected font metrics structure to the Font Attribute Table (*Fattrs-Match* and *Fattrs-FaceName*).
- 6) Issue the GpiCreateLogFont call.

The font metrics returned by the GpiQueryFonts call contain three fields that help define the characteristics of each font. These fields, *Fmetrs-Type*, *Fmetrs-Defn* and *Fmetrs-Selection*, contain the following information:

FONT METRICS INFORMATION

Fmetrics-Type	Hex Value	Definition
FM-TYPE-FIXED	(0x0001)	All characters have the same width.
FM-TYPE-LICENSED	(0x0002)	This font is a licensed font.
FM-TYPE-KERNING	(0x0004)	Font contains kerning information.
FM-TYPE-DBCS	(0x0010)	Font uses a double-byte code page.
FM-TYPE-MBCS	(0x0018)	Font uses a mixed-byte code page.
FM-TYPE-64K	(0x8000)	Font is larger than 64K bytes.
Fmetrics-Defn	Hex Value	Definition
FM-DEFN-OUTLINE	(0x0001)	This is a vector (outline) font.
FM-DEFN-GENERIC	(0x8000)	Font format can be used by Gpi.
Fmetrics-Selection	Hex Values	Definition
FM-SEL-ITALIC	(0x0001)	Font contains italic characters.
FM-SEL-UNDERSCORE	(0x0002)	Font characters are underscored.
FM-SEL-NEGATIVE	(0x0004)	Characters have foreground / background colors reversed.
FM-SEL-OUTLINE	(0x0008)	Characters are hollow outlines.
FM-SEL-STRIKEOUT	(0x0010)	Characters have a line running through them.
FM-SEL-BOLD	(0x0020)	Characters have a bold weight.

Storing all the metrics returned by the GpiQueryFonts call for a specific font face can consume a large amount of Local-Storage - several thousand bytes for some faces. This is more stack space than many COBOL programs can afford to allocate. To overcome this, programs may obtain additional memory for storage of the returned metrics using the DosAllocSeg call for Version 1.3 and the DosAllocMem call for Version 2.0. See Chapter 16 for information on issuing Dos calls from within a COBOL program. If your program uses stack memory to hold the returned fonts, be sure to increase the stack size specification in the Linker Definition file before linking the program. For the sample program, STACKSIZE must be set to 32768 (32K) before this chapter's calls can be implemented.

If the program does not contain enough stack space to retrieve all the metrics for a font, the number of returned metrics can be limited by placing the maximum number of metrics to be returned in the variable pointed to by parameter four of the GpiQueryFonts call. If the GpiQueryFonts call is made with a limiting number, the return code should

be checked for the number of fonts meeting the search criteria that could not be returned. If the return code is greater than 0, the search criteria should be narrowed and query issued again.

The sample program requires a monospaced font (all characters have the same width) with the point size and weight depending upon the density of the display screen. If the screen width is 640 pels or less, a 10 point standard Courier font is used (*Fattr-
MaxBaseLineExt = 12* and *Fattr-AveCharWidth = 9*). If the screen width is greater than 640 pels, then a 12 point bold Courier is used (*Fattr-Selection = Fattr-Sel-Bold*, *Fattr-
MaxBaseLineExt = 15* and *Fattr-AveCharWidth = 12*).

But the logic that is used to select the font allows for some flexibility. The sample program requires a monospaced font so it queries PM for all available Courier fonts. From the list of returned fonts, the program tries to select a point size and character width that are best for the display in use. Here is the logic used by the sample program to select the best font for display of the PM calls.

From the list of Courier fonts -

Select a font with specific point size, maximum base line extent and average character width based upon display in use.

If no match made, select a font based on point size only.

If no match made, select any Courier font.

If no match made, use the default font.

Establishing The Current Font

Text can only be drawn into the presentation space using the current font. When a presentation space is created, the System Proportional image font is established as the current font. If the program takes no action to change the current font, then this default font is used for all text drawn into the presentation space. If the program wishes to use a font other than the default font, it must perform the following steps.

- Create a logical font, the rendering of the required font, using the *GpiCreateLogFont* call. Creating the logical font rasterizes an outline font, placing a bit mapped image of each character, in the requested font, into the presentation space. For image fonts, *GpiCreateLogFont* loads the correct font image table into the presentation space. All public fonts, plus any private fonts previously loaded, are available for use with the *GpiCreateLogFont* call. Up to 254 logical fonts may be loaded at any one time.

```

01  FontMetrics.
05  Fmetrs-Table.
10  Fmetrs-FamilyName      pic  X(32) .
10  Fmetrs-FaceName        pic  X(32) .
10  Fmetrs-Registry         pic  9(4)   comp-5.
10  Fmetrs-CodePage         pic  9(4)   comp-5.
10  Fmetrs-EmHeight         pic  s9(9)  comp-5.
10  Fmetrs-XHeight          pic  s9(9)  comp-5.
10  Fmetrs-MaxAscender      pic  s9(9)  comp-5.
10  Fmetrs-MaxDescender      pic  s9(9)  comp-5.
10  Fmetrs-LowerCaseAscent    pic  s9(9)  comp-5.
10  Fmetrs-LowerCaseDescent   pic  s9(9)  comp-5.
10  Fmetrs-InternalLeading     pic  s9(9)  comp-5.
10  Fmetrs-ExternalLeading      pic  s9(9)  comp-5.
10  Fmetrs-AveCharWidth       pic  s9(9)  comp-5.
10  Fmetrs-MaxCharInc         pic  s9(9)  comp-5.
10  Fmetrs-EmInc              pic  s9(9)  comp-5.
10  Fmetrs-MaxBaseLineExt      pic  s9(9)  comp-5.
10  Fmetrs-CharSlope           pic  s9(4)  comp-5.
10  Fmetrs-InlineDir            pic  s9(4)  comp-5.
10  Fmetrs-Charrot              pic  s9(4)  comp-5.
10  Fmetrs-WeightClass          pic  9(4)   comp-5.
10  Fmetrs-WidthClass            pic  9(4)   comp-5.
10  Fmetrs-XDeviceRes          pic  s9(4)  comp-5.
10  Fmetrs-YDeviceRes          pic  s9(4)  comp-5.
10  Fmetrs-FirstChar             pic  s9(4)  comp-5.
10  Fmetrs-LastChar              pic  s9(4)  comp-5.
10  Fmetrs-DefaultChar           pic  s9(4)  comp-5.
10  Fmetrs-BreakChar             pic  s9(4)  comp-5.
10  Fmetrs-NominalPointSize       pic  s9(4)  comp-5.
10  Fmetrs-MinimumPointSize       pic  s9(4)  comp-5.
10  Fmetrs-MaximumPointSize       pic  s9(4)  comp-5.
10  Fmetrs-Type                  pic  9(4)   comp-5.
10  Fmetrs-Defn                  pic  9(4)   comp-5.
10  Fmetrs-Selection               pic  9(4)   comp-5.
10  Fmetrs-Capabilities            pic  9(4)   comp-5.
10  Fmetrs-SubscriptXSize          pic  s9(9)  comp-5.
10  Fmetrs-SubscriptYSize          pic  s9(9)  comp-5.
10  Fmetrs-SubscriptXOffset         pic  s9(9)  comp-5.
10  Fmetrs-SubscriptYOffset         pic  s9(9)  comp-5.
10  Fmetrs-SuperscriptXSize         pic  s9(9)  comp-5.
10  Fmetrs-SuperscriptYSize         pic  s9(9)  comp-5.

```

Figure 12-3 The font metrics structure - Part A

10 Fmetrs-SubscriptXOffset	pic s9(9)	comp-5.
10 Fmetrs-SubscriptYOffset	pic s9(9)	comp-5.
10 Fmetrs-SuperscriptXSize	pic s9(9)	comp-5.
10 Fmetrs-SuperscriptYSize	pic s9(9)	comp-5.
10 Fmetrs-SuperscriptXOffset	pic s9(9)	comp-5.
10 Fmetrs-SuperscriptYOffset	pic s9(9)	comp-5.
10 Fmetrs-UnderscoreSize	pic s9(9)	comp-5.
10 Fmetrs-UnderscorePosition	pic s9(9)	comp-5.
10 Fmetrs-StrikeoutSize	pic s9(9)	comp-5.
10 Fmetrs-StrikeoutPosition	pic s9(9)	comp-5.
10 Fmetrs-KerningPairs	pic s9(4)	comp-5.
10 Fmetrs-FamilyClass	pic s9(4)	comp-5.
10 Fmetrs-Match	pic s9(9)	comp-5.

Figure 12-3 The font metrics structure - Part B

- Make the new logical font the current font by issuing the GpiSetCharSet call.
- Obtain the font metrics of the current font using the GpiQueryFontMetrics call. Before text can be drawn, the characteristics of the font must be made available to the program so that the text may be correctly positioned within the presentation space. There are more than forty different pieces of information within a font metrics that can affect the correct placement of characters within the presentation space. Figure 12-3 shows the metrics information available for each font. Figure 12-4 shows how some of the most commonly used metrics information relates to the text being drawn.
- Draw the text into the presentation space using one of several Gpi drawing commands. The graphical subsystem replaces each of the characters received with its bit map image taken from the current font's font table and places it in the correct position in the presentation space.
- Upon completion of the drawing task, if the presentation space is to be retained, replace the current font with the prior font using the GpiSetCharSet call, specifying the Local Character-set ID of the prior font or the default font.
- If the font is no longer needed, delete the logical font using the GpiDeleteSetId call. Each font consumes a considerable amount of space within the presentation space.

For additional information about fonts and their metrics, please review the related chapters in the *Presentation Manager Programming References*. These references are part of the OS/2 Developer's Toolkit.

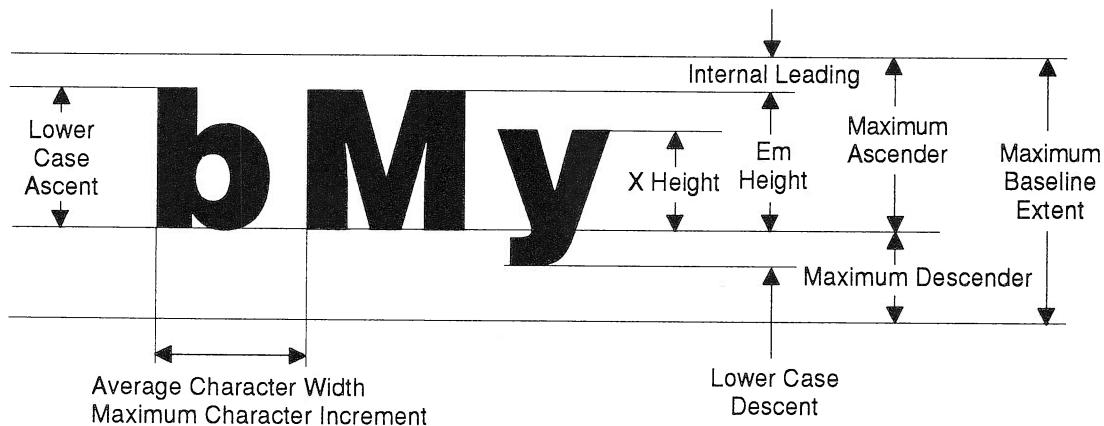


Figure 12-4 Selected font metric measurements

Testing For The Required Font

Fonts are an installable feature of the OS/2 operating system. As such, the required font may not be installed on the computer used to run your program. So, when a program requires an installable feature, like fonts, the program must test for the availability of the required installable features during startup processing. Testing for the Courier font should be added in this chapter along with the font processing, but checking for available fonts involves processing the .INI files, and .INI file processing is covered in Chapter 14. Until the proper checking is added in Chapter 14, be sure that the Courier image font is available before running this chapter's sample program.

Retrieving The Data To Be Displayed

The objective of this chapter is to display the PM call selected by the user from the PM Call dialog. The actual code to be displayed is contained in the PMCALLS.TXT file. The file contains the call's text stored in the exact format for display when the Courier monospaced font is used. The format of this file is shown in Figure 12-5. Note that the File-Control variable *UT-S-SourceList* contains a fully qualified path statement for the PMCALLS.TXT file. You will want to change this path statement to match your environment. I acknowledge that coding the path of the PMCALLS.TXT file into the program is not a desirable procedure. However, use of the OS/2 Kernel calls required to dynamically locate this file will not be discussed until a later chapter. So, for the present, code this string to match your environment.

```
WinDestroyWindow          ****
call OS2API 'WinDestroyWindow' using
    by value hwndFrame
    returning ReturnData

WinDlgBox                ****
call OS2API 'WinDlgBox' using
    by value      HWND-DESKTOP
    by value      hwndClient
    by value      WindowProc
    by value      ShortNull
    by value      ID-PMCallDlgBox
    by reference  PMCall
    returning     PMCReturn

WinDismissDlg             ****
call OS2API 'WinDismissDlg' using
    by value      hwnd
    by value      DPMCReturn

WinDispatchMsg            ****
call OS2API 'WinDispatchMsg' using
    by value      hab
    by reference  QMSG
    returning     ReturnData

    •
    •
    •
    •

WinWindowFromID           ****
call OS2API 'WinWindowFromID' using
    by value      hwndFrame
    by value      FID-MENU
    returning     hwndMenu
```

Figure 12-5 The structure of the PMCALS.TXT file

To access this file, a disk processing routine must be added after the return from the PMCall dialog. If the dialog resulted in a call being selected, the program must remove the null termination character from the PM call string, then locate the call in the PMCALLS.TXT file. It is not my intention to spend time on the file processing code added to the sample program in this chapter. Nothing in this routine is unique to PM programming, and most of you are proficient at writing disk processing routines. The routine used in the sample program begins on source line 014000.

Modifications To The Sample Program

When graphics are to be placed into a window and the presentation space is retained from one drawing to the next, the presentation space must be cleared of existing text before any new text is drawn. The sample program does not have this requirement, as a new presentation space is acquired for each refresh of the Source Code window. However, you should know how to clear the presentation space using the GpiErase call.

The GpiErase call is used to clear the presentation space of graphics and reset the background to its default color. GpiErase requires the handle of the presentation space as parameter one and the variable that is to receive the call's return code as parameter two. Here is the format of the GpiErase call:

```
call OS2API 'GpiErase' using
            by value hps
            returning ReturnData
```

In the sample program, after the presentation space is filled with the current window background color, PMCallLineCt is checked to see if any text is to be drawn in the presentation space. If text is to be drawn, *300-Screen-Write* is performed. If no text is to be drawn (*PMCallLineCt = 0*), the *Window-Paint* routine completes by releasing the presentation space.

At the time the text is to be drawn, the exact font to be used must be selected from the available public and private fonts. For the sample program, a 10 point Courier font is used for VGA type displays (640 x 480 pels) and a 12 point Courier bold font for 8514/A and XGA type displays (1024 x 768 pels).

With the size of the current display screen returned via the WinQuerySysValue call, the Courier font metrics can be parsed to determine the font that best meets the program's requirements. The sample program preallocates a Font Metrics Table of 13 entries, the exact number of Courier image fonts available with the base OS/2 system, then issues the GpiQueryFonts call without a limit on the number of metrics.

Coding The GpiQueryFonts Call

The GpiQueryFonts call requires the handle of the presentation space as parameter one. Parameter two defines the category of fonts that may be searched by PM. The two categories are QF-PUBLIC and QF-PRIVATE. As the terms indicate, they define which of the two font classes to search. If both types of fonts are to be searched, then these two values must be ORed together.

Parameter three is a pointer to the font face name. If a face name is supplied, PM will restrict the search to this face name. If a null value is supplied, then all faces will be enumerated. Parameter four is a long integer that contains a limit on the number of font metrics that may be returned. If this variable is 0, the number of fonts that meet the search criteria will be returned by the call. If this variable contains a value greater than 0, then the first *n* number of fonts that meet the search requirements will be returned, where *n* is the number contained in this variable.

Parameter five contains the length of each metric table entry. Depending upon the amount of metric information needed by the program, the length of each table entry can be limited by this parameter. The full metrics table is 208 bytes long. Parameter six is a pointer to the start of the metrics table area. The size of the memory set aside for the returned font metrics information (pointed to by parameter 6) must be equal to the number of fonts returned (parameter 4) times the size of each font (parameter 5).

The last parameter is the variable that is to receive the call's return code. Here is the GpiQueryFonts call used in the sample program to enumerate the Courier public fonts:

```

015200      Call OS2API 'GpiQueryFonts' using
015210              by value      hps
015220              by value      QF-PUBLIC
015230              by reference Fattrr-Name
015240              by reference LongWork
015250              by value      MetricsLength
015260              by reference FontMetrics
015270              returning    ReturnData

```

Parsing The Font Metrics File

After clearing the Font Attribute Table to zeros, a perform is executed to parse the font metrics returned by the GpiQueryFonts call. This parsing routine examines each font

metric using criteria established by the program, and selects the best possible font from those returned by PM. There is no standard criteria for font selection that is applicable to every program. The desired font should be determined when the application is designed and the logic added to parse for that font or an acceptable alternative. Whatever criteria is used to select the font, once it is found, the match number along with the font face name must be moved from the font's metrics table to the Font Attributes Table. These two entries are all that is needed to ensure that the desired font is created.

Here is the code used in the sample program to parse the metrics tables and select the desired font. This code follows the logic shown in the prior section, *Selecting The Desired Font*.

```

If SizeWide > 640
    If Fmetrs-NominalPointSize(Indx) = 120
        Move FATTR-SEL-BOLD to Fattrss-Selection
        If Fattrss-Match = 0
            Move Fmetrs-Match(Indx) to Fattrss-Match
            Move Fmetrs-FaceName to Fattrss-FaceName
        End-if
        If (Fmetrs-MaxBaseLineExt(Indx) = 15) and
            (Fmetrs-AveCharWidth(Indx) = 12)
            Move Fmetrs-Match(Indx) to Fattrss-Match
            Compute Indx = LongWork + 1
        End-if
    End-if
Else
    If Fmetrs-NominalPointSize(Indx) = 100
        If Fattrss-Match = 0
            Move Fmetrs-Match(Indx) to Fattrss-Match
            Move Fmetrs-FaceName to Fattrss-FaceName
        End-if
        If (Fmetrs-MaxBaseLineExt(Indx) = 12) and
            (Fmetrs-AveCharWidth(Indx) = 9)
            Move Fmetrs-Match(Indx) to Fattrss-Match
            Compute Indx = LongWork + 1
        End-if
    End-if
End-if.

```

Following the selection of the font to be used, the Font Attribute Table must be completed. The length of the table (*Fattrss-Length*) must be set to indicate how much of the table contains valid information. The sample program uses the last table entry so the table length must be set to its maximum 56-byte length.

Further restrictions on the font to be created may be set using the FATTRS flags. These flags set specific font properties, and are passed in the *Fattr-Selection*, *Fattr-Type* and *Fattr-FontUse* fields of the Font Attribute Table. The flags in these three fields may be combined in whatever combination is required to achieve the desired font properties. Here are the flags used in the three fields:

FONT ATTRIBUTE INFORMATION

Fattr-Selection	Hex Value	Definition
FATTR-SEL-ITALIC	(0x0001)	Generate an Italics font.
FATTR-SEL-UNDERSCORE	(0x0002)	Underscore the characters.
FATTR-SEL-OUTLINE	(0x0008)	Do not fill the character outlines. This function will improve the drawing times for outline fonts.
FATTR-SEL-STRIKEOUT	(0x0010)	Generate characters with a horizontal line through them.
Fattr-FontUse	Hex Value	Definition
FATTR-FONTUSE-NOMIX	(0x0002)	The text will not be intermixed with other graphics.
FATTR-FONTUSE-OUTLINE	(0x0004)	Use an outline (vector) font.
FATTR-FONTUSE-TRANSFORMABLE	(0x0008)	Characters may be scaled, rotated or sheared.
FATTR-SEL-BOLD	(0x0020)	Generate a bold font.
Fattr-Type	Hex Value	Definition
FATTR-TYPE-KERNING	(0x0004)	Enable PostScript kerning.
FATTR-TYPE-MBCS	(0x0008)	Multi-byte character set required.
FATTR-TYPE-DBCS	(0x0010)	Double-byte character set required.
FATTR-TYPE-ANTIALISED	(0x0020)	Antialised font required. Only valid if device driver support available.

The sample program uses the *FATTR-FONTUSE-NOMIX* flag to indicate that text and graphics will not be mixed in the same presentation space, the *FATTR-FONTUSE-OUTLINE* flag to request an image font (the flag is not set), and the *FATTR-SEL-BOLD* flag to request a bold font when using displays greater than 640 pels wide.

Courier Image Fonts Available With OS/2								
Nominal Point Size	* Maximum Base Line Extent	XHeight	Average Character Width	EmHeight	Maximum Ascender	Maximum Descender	Weight & Width Class	‡ Match Number
80 [†]	10	5	8	8	9	1	medium	3
100	12	6	9	10	10	2	medium	4
120	15	7	12	16	12	3	medium	5
80	13	6	8	10	11	2	medium	6
100	16	7	9	13	13	3	medium	7
120	20	9	12	16	16	4	medium	8
80	10	5	10	8	9	1	medium	9
100	12	6	12	10	10	2	medium	10
120	12	6	12	12	10	2	medium	11
80	7	4	8	5	6	1	medium	12
100	8	5	9	6	7	1	medium	13
120	10	5	12	8	9	1	medium	14
120	18	6	6	16	13	4	medium	68

Values are in world coordinates unless indicated otherwise

[†] Values returned in decipoins (1 point = 10 decipoins)

* For image fonts this field contains the height of the font

‡ Negative match numbers indicate device fonts, positive match numbers indicate public fonts

Figure 12-6 Courier fonts available with OS/2

Here is how the Font Attribute Table is completed in the sample program. Remember, the face name and match number were placed into the Font Attribute Table during the parsing routine.

```

015330      If Fattrs-Match > 0
015340          Move 56 to Fattrs-Length
015350          Move FATTR-FONTUSE-NOMIX to Fattrs-FontUse
015360          Move 1 to LCID-PMC

```

As part of the GpiCreateLogFont call, an ID, called a local font ID or LCID, must be associated with the logical font. This ID is similar to a window handle in that it becomes the primary reference to the font for all font-related calls while the font is in the presentation space. Any value in the range 1 to 254, not already assigned to a logical font, may be used. For the sample program an ID of 1 is used.

Coding The GpiCreateLogFont Call

The GpiCreateLogFont call uses the Font Attribute Table to pass the program's font requirements to PM. PM then selects the font that best fits the requirements, rasterizes the font if necessary, assignes a local character-set identifier (LCID) and stores the font in the application's presentation space. To code the GpiCreateLogFont call, place the handle of the presentation space in parameter one. Parameter two is a pointer to the string used to describe this font when it is passed to another system as part of an interchange file. Any null-terminated descriptive text string up to 8 characters long may be used. Parameter three is the variable holding the ID to be assigned to this font. Parameter four is a pointer to the Font Attribute Table and parameter five is the variable that will receive the call's return code. Here is the GpiCreateLogFont call used in the sample program:

```
015370      Call OS2API 'GpiCreateLogFont' using
015380              by value      hps
015390              by reference Fattr -Name
015400              by value      LCID -PMC
015410              by reference Attrs
015420          returning     ReturnData
```

Coding The GpiSetCharSet Call

Before a newly created logical font can be used, it must be established as the current font using the GpiSetCharSet call. The GpiSetCharSet call replaces the current font with the font having the specified local character-set ID. There are usually multiple logical fonts in a presentation space at any time, but only the current font will be used with the next drawing command. Creating a document or display with multiple fonts becomes a process of changing the current font so that the correct font for the next character to be drawn is the current font.

The GpiSetCharSet call requires the handle of the presentation space as parameter one, the local character-set ID established in the GpiCreateLogFont call as parameter two, and the variable that will receive the call's return code as parameter three. Here is the GpiSetCharSet call used in the sample program:

```
015440      Call OS2API 'GpiSetCharSet' using
015450          by value    hps
015460          by value    LCID-PMC
015470          returning ReturnData
```

Coding The GpiQueryFontMetrics Call

With the desired font established as the current font, the text can now be drawn into the presentation space. However, there are no fixed rows and columns for the placement of graphical text. The program must determine to the exact pel where each line of text is to be placed and account for such things as character spacing, character ascenders and descenders, internal and external leading and the character's slope and shear. These values are different for each font variation, so the program must retrieve the metrics information about the current font before the text can be correctly positioned in the presentation space.

The font metrics for the current font are obtained by issuing the GpiQueryFontMetrics call. This call returns the same structure as is returned by the GpiQueryFonts call, but only for the current font. The actual number of bytes returned is controlled by the value placed in parameter two of the call. The full metric structure is 208 bytes long, but depending upon which data the program requires, less than 208 bytes may be returned.

To code the GpiQueryFontMetrics call, place the handle of the presentation space in parameter one. Parameter two is a long integer containing the maximum amount of metrics data that may be returned by this call. Parameter three is a pointer to the Font Metrics structure. As this call returns metrics information only for the current font, the local font ID is not specified. Here is the GpiQueryFontMetrics call used in the sample program:

```
015500      Call OS2API 'GpiQueryFontMetrics' using
015510          by value    hps
015520          by value    MetricsLength
015530          by reference FontMetrics
015540          returning   ReturnData
```

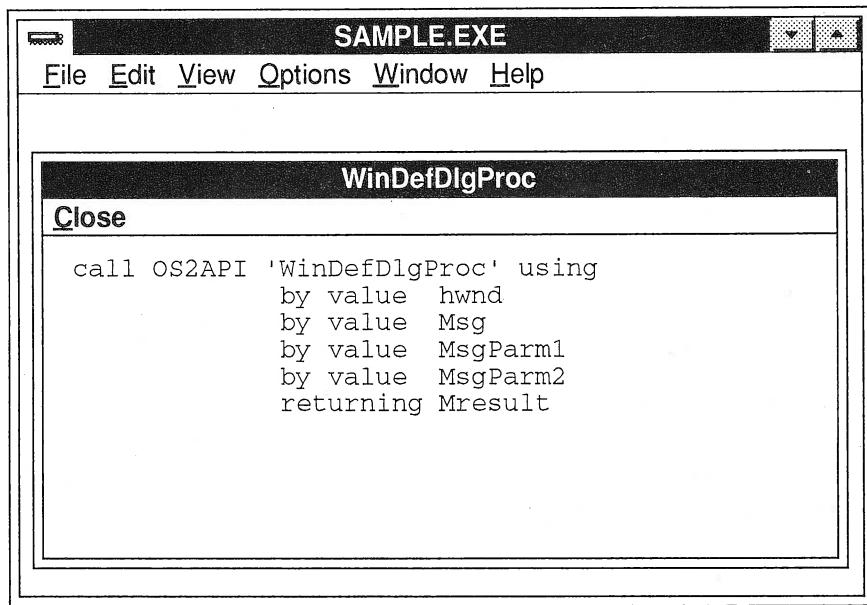


Figure 12-7 The Source Code window displaying a PM call (Version 1.3)

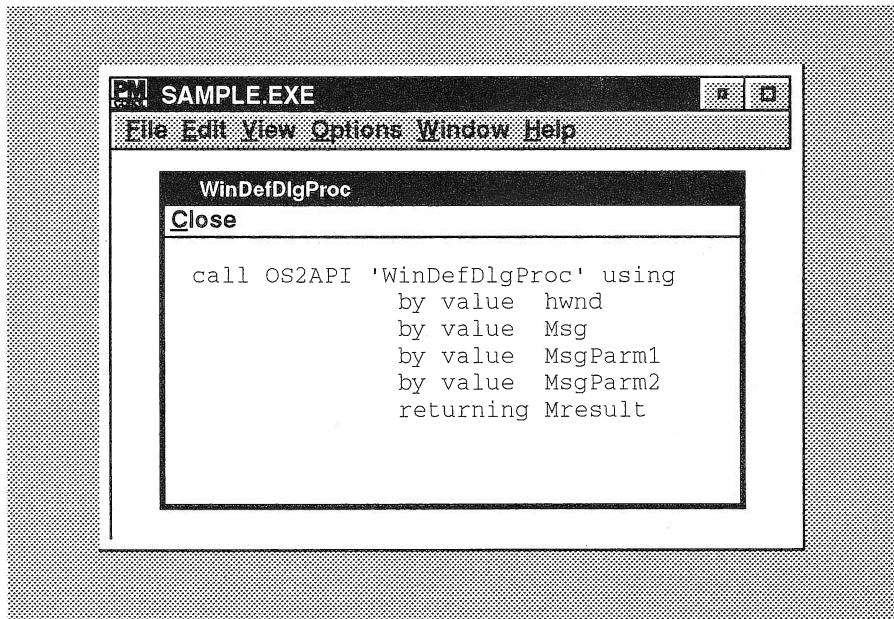


Figure 12-8 The Source Code window displaying a PM call (Version 2.0)

Coding The GpiCharStringAt Call

There are two categories of string drawing commands available, depending upon the amount of control to be exercised over individual character placement. For drawing with control over the placement of each character, use GpiCharStringPosAt if the starting position is to be passed, or GpiCharStringPos if drawing is to start at the presentation space's current x,y coordinates. These calls should be used for critical character placement, such as text justification.

For drawing with no control over the individual characters, use GpiCharStringAt if the starting position is to be passed, or GpiCharString if drawing is to start at the presentation space's current x,y coordinates. These calls should be used when individual character placement is not critical. For the sample program, I am using the Courier monospaced font so precise character control within each string is not required. As a result, the GpiCharStringAt call is used.

For the calls that use the current x,y presentation space coordinates as the starting point for text placement, the GpiMove call allows new x,y coordinates to be set before issuing the string drawing call.

Because the output is composed of a variable number of lines the actual GpiCharStringAt call is placed within a perform with a test to limit the number of lines written to the value in *PMCallLineCt*. After setting the x coordinate and the maximum length of each line, here is the perform statement used in the sample program.

```

015660      Move 10 to x-coord of rect
015670      Move 60 to LineLength.
015680      Perform 320-Screen-Out
015690          varying indx from 1 by 1 until indx > PMCallLineCt

```

The *320-Screen-out* routine positions each line using a very simple formula. A more sophisticated formula must be used where exact text placement is a requirement. The sample program places each line of text the length of the maximum base line extent plus the external leading below the prior line (*Fmetrs-MaxBaseLineExt + Fmetrs-ExternalLeading*), with the first line an additional 10 pels below the top of the window (*YTOP-10*). See Figure 12-9.

The maximum baseline extent is the sum of the maximum ascender and maximum descender values for the current font (see Figure 12-4). Depending upon the type of text being drawn and the desired output, any one or a combination of the metric values may be used to position the text. The advantage of using the metrics labels within the text

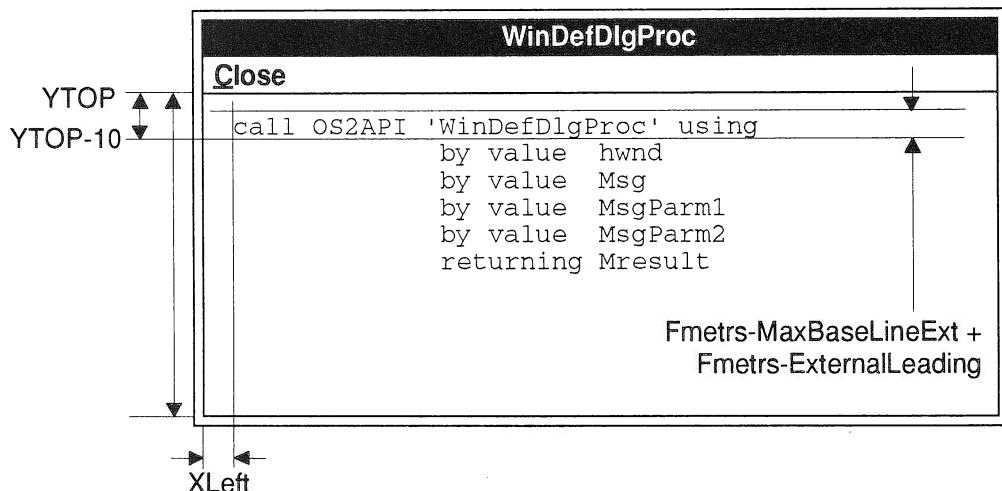


Figure 12-9 Text positioning measurements

placement formula is the automatic adjustment of the values to reflect the current font's metrics.

To code the `GpiCharStringAt` call, place the handle of the presentation space in parameter one. Parameter two is a pointer to the structure holding the string's starting x,y coordinates, defined in world coordinates. The x,y coordinates are relative to the lower left corner of the window. Parameter three is the count of the number of characters in the text string. For the sample program, each of the output lines is fixed at 60 characters. Parameter four is a pointer to the current line of text. For the sample program, the text string pointer points to the current line of text within an array. The last parameter is the variable that will receive the call's return code. Here is the `GpiCharStringAt` call used in the sample program:

```

016230      Compute y-coord = (YTOP - 10) -
016240            (Indx * (Fmetrics-MaxBaseLineExt(Indx) +
016250                  Fmetrics-ExternalLeading(Indx)))
016260
016270      Call OS2API   'GpiCharStringAt' using
016280            by value      hps
016290            by reference rect
016300            by value      LongWork
016310            by reference PMCallLines(Indx)
016320            returning     ReturnData

```

Resetting The Current Font

Where the presentation space is to be retained for a period of time or where multiple routines will use the same presentation space, the font in use prior to the current routine should be restored as the current font. Depending upon the length of time before the logical font is to be reused, it may need to be destroyed and then recreated to create additional space within the presentation space for other functions. When the presentation space is destroyed after the drawing is completed, as in the sample program, the resetting of the current logical font and the destruction of the logical font are not required. Destroying the presentation space will destroy the logical fonts. I have included these two calls in the sample program only as examples of their use.

Resetting the logical font uses the same GpiSetCharSet call used to establish the current font. Parameter two must contain the LCID of the font to be reestablished as the current font, or as in the sample program, the presentation space default font (*LCID-Default*). Here is the GpiSetCharSet call to reestablish the default System Proportional font as the current font:

```

015710      Call OS2API 'GpiSetCharSet' using
015720                  by value hps
015730                  by value LCID-Default
015740                  returning ReturnData

```

Coding The GpiDeleteSetId Call

This call deletes the logical font with the LCID passed in parameter two from the presentation space. Deletion of a logical font normally occurs when multiple fonts are required within a single presentation space and there is insufficient room for all fonts, or space taken by logical fonts is needed for other presentation space drawing. There is no need to delete logical fonts after every use and certainly not if they are being used on a regular basis.

The GpiDeleteSetId requires only the handle of the presentation space as parameter one, the local character-set ID to be deleted as parameter two and the call's return code variable as parameter three. For the sample program, *LCID-PMC* contains the assigned ID of the Courier font used to display the PM call and now to be deleted. Here is the GpiDeleteSetId call used in the sample program:

```

015760      Call OS2API 'GpiDeleteSetId' using
015770          by value hps
015780          by value LCID-PMC
015790          returning ReturnData.

```

Changing The Look Of Other Windows

This chapter has concentrated on font manipulation through the creation of logical fonts within a window's presentation space. But there are times when you may need to change the font, color or border of a control window that is not within your presentation space. Dialog window controls are a good example. For these situations, the WinSetPresParam call supports the manipulation of window elements for windows not under direct program control.

WinSetPresParam associates a presentation parameter attribute with the window element's handle. If the attribute already exists, PM changes the attribute to the value contained in the WinSetPresParam call. If the attribute does not exist, PM adds the attribute to the window's presentation parameters. WinSetPresParam allows the manipulation of a window's foreground and background color, color highlighting, border color and font face name and size. The following table shows the presentation parameters that may be changed along with their defined values.

Window Presentation Parameter	Decimal Value
PP-ForeGroundColor	1
PP-ForeGroundColorIndex	2
PP-BackGroundColor	3
PP-BackGroundColorIndex	4
PP-HiLiteForeGroundColor	5
PP-HiLiteForeGroundColorIndex	6
PP-HiLiteBackGroundColor	7
PP-HiLiteBackGroundColorIndex	8
PP-DisableForeGroundColor	9
PP-DisableForeGroundColorIndex	10
PP-DisableBackGroundColor	11
PP-DisableBackGroundColorIndex	12
PP-BorderColor	13
PP-BorderColorIndex	14
PP-FontNameSize	15
PP-FontHandle	16

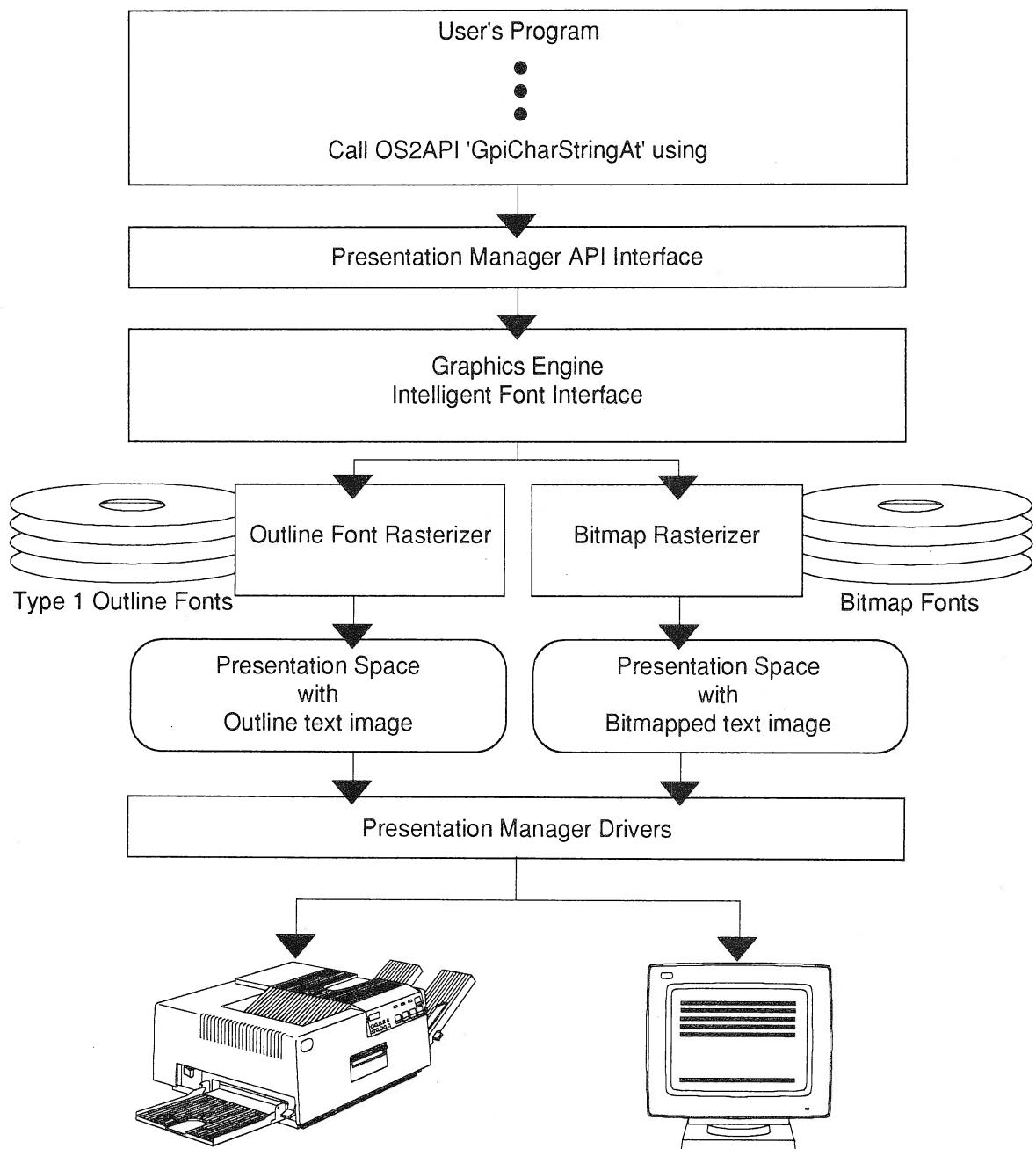


Figure 12-10 The text conversion process

The string value used for PP-FontNameSize is composed of two parts, separated by a period. The first half of the string is the font point size. This is followed by a period, then the second half of the string, the image font face name. Outline fonts are not supported by the WinSetPresParam call. The string *12.Courier* is the proper PP-FontNameSize entry for a 12 point Courier font.

The font attribute values are case sensitive and must be coded correctly or the font will not be found. The attribute values for the OS/2 supplied image font face names are as follows.

Font Face Name	Presentation Parameter
Courier Font	Courier
Helv Font	Helv
System Monospaced Font	SysMono
Times Roman Font	Times

Coding The WinSetPresParam Call

The WinSetPresParam call requires the handle of the window to be modified as parameter one. Parameter two contains the system attribute identifier; to change the font or font face name, use the PP-FontNameSize parameter. Parameter three is the length of the string value specified in parameter four, including the null termination character. Parameter four contains a pointer to the presentation parameter string. Parameter five is the variable that will contain the call's return code. Here is an example of how to code the WinSetPresParam call to change the text displayed on the title bar of *hwndWindow* to a 12 point Courier font.

```
01 FontString  pic x(11) value "12.Courier" & x"00".
```

```
Move 11 to ULongWork
Call OS2API 'WinSetPresParam' using
      by value      hwndSource
      by value      PP-FONTNAMESIZE
      by value      ULongWork
      by reference  FontString
      returning     ReturnData
```

Chapter 13

Buttons, Boxes And Entry Fields

This chapter's discussion of radio buttons, check boxes and entry fields together with the prior chapter's discussion of list boxes and push buttons, completes the introduction to dialogs, dialog processing and dialog controls. There are additional dialog controls; such as combo boxes, spin buttons and multiline entry fields, and as part of OS/2 Version 2.0, container, notebook, slider and value set controls. However, the use of these controls does not differ materially from the dialog controls I have discussed in this book. For a complete discussion of all dialog controls, see the *Presentation Manager Programming References*, part of the OS/2 Developer's Toolkit.

There are six dialog controls that allow the user to make a selection. Three - radio buttons, spin buttons, and push buttons - offer the user a single selection from among a group or list of items, check boxes offer the user multiple selections from among a group or list of items, while two - containers and value sets - can be programmed to support either type of selection. Radio buttons, spin buttons and check boxes allow the user to make repetitive selections before any action is taken. Push buttons allow the user only a single selection, with action taken as soon as the selection is made. Containers and value sets can be programmed to react in either way.

This chapter will discuss two of these selection controls, radio buttons and check boxes.

Radio Buttons

Radio buttons, referred to as single choice controls, present a fixed set of mutually exclusive choices to the user. The user may select only one radio button at a time from the available group of buttons. Like radio buttons on a car radio, one of the buttons must always be selected and when another selection is made, the prior selection must be

cleared. The user may make as many selections as desired while the dialog is active; the last button selected when the dialog completes becomes the selected control. The Window Colors controls shown in Figures 13-1 and 13-2 are examples of radio buttons.

A radio button is displayed as a small circle followed by a string of text that describes the choice represented by the button. Clicking on either the circle or the text will select the button and cause the circle to be filled with a black dot. When the user selects a radio button, PM sends a WM-CONTROL message to the program with the BN-CLICKED message identification and the resource ID of the button selected.

There are two styles of radio buttons, regular radio buttons with the style of BS-RADIOBUTTON and auto-radio buttons with the style of BS-AUTORADIOBUTTON.

Regular radio buttons offer the standard look, feel and function of this style of button control, but require that the program manage the state of the buttons. In response to the BN-CLICKED message, the program must send a BM-QUERYCHECK message to the dialog to determine the state of the button. If the button was selected, then the program issues the BM-SETCHECK message with message parameter one set to true to cause the button to be painted. If the button was cleared, then the program issues the BM-SETCHECK message with message parameter one set to false to cause the button to be cleared. In addition, the program must clear the previously selected button by sending a BM-SETCHECK message to that button.

Auto-radio buttons offer the same look, feel and function as regular radio buttons, but the dialog procedure automatically performs the button state painting for both the selected and deselected buttons. As with regular radio buttons, the program is notified when a button is selected via a BN-CLICKED message.

Radio button style selection is made when the dialog is created.

Check Boxes

Check boxes, referred to as multiple choice controls, allow the user to select more than one item from a fixed group of choices. Check boxes serve the same function as radio buttons, but allow the simultaneous selection of any number of the items within the group. The user may make as many selections as desired while the dialog is active; the boxes selected when the dialog completes become the selected controls.

A check box consists of a small square followed by a string of text that describes the choice represented by the box. Clicking on either the box or the text will select the item and cause an X (Version 1.3) or a check mark (Version 2.0) to be placed inside the square. When a check box is cleared, the X or check mark is removed from the square. When the

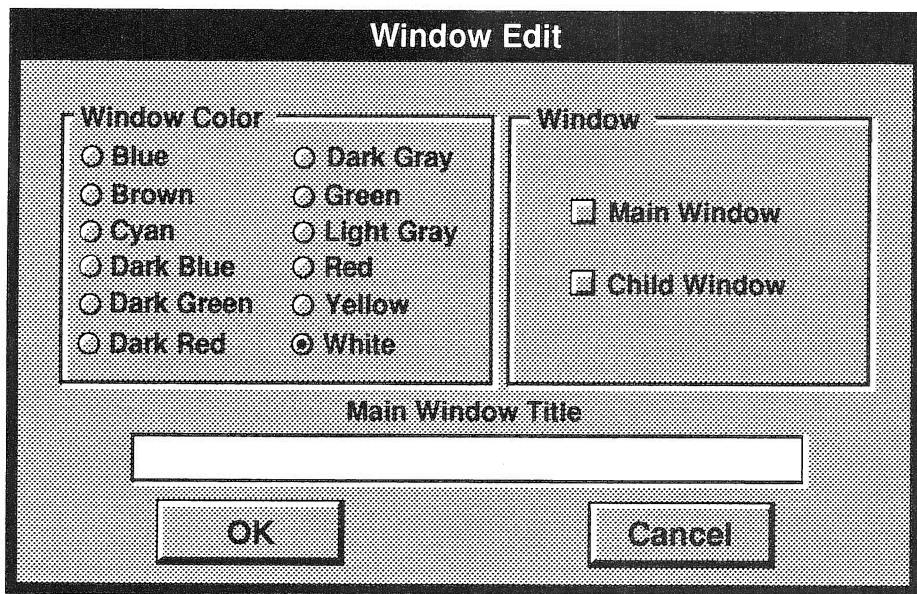


Figure 13-1 Window Edit dialog box (Version 2.0)

user selects a check box, PM sends a WM-CONTROL message to the program with the BN-CLICKED message identification and the resource ID of the box selected.

As with radio buttons, there are two styles of check boxes, regular check boxes with the style of BS-CHECKBOX and auto-check boxes with the style of BS-AUTOCHECKBOX.

Regular check boxes offer the look, feel and function of this style of control, but require the program to manage the state of the boxes. In response to the BN-CLICKED message, the program must send a BM-QUERYCHECK message to the dialog to determine the state of the box. If the box was selected, then the program issues the BM-SETCHECK message with message parameter one set to true to place an X or check mark in the box. If the box was cleared, then the program issues the BM-SETCHECK message with message parameter one set to false to clear the box. But unlike radio buttons, no other box must be cleared as the result of a box selection message, since selecting one box does not imply the deselecting of another box.

Auto-check boxes offer the same look, feel and function of check boxes, but the dialog procedure automatically performs the state painting of the box. As with radio buttons, the program is notified when a check box is selected or cleared via a BN-CLICKED message.

As with radio buttons, check box style selection is made when the dialog is created.

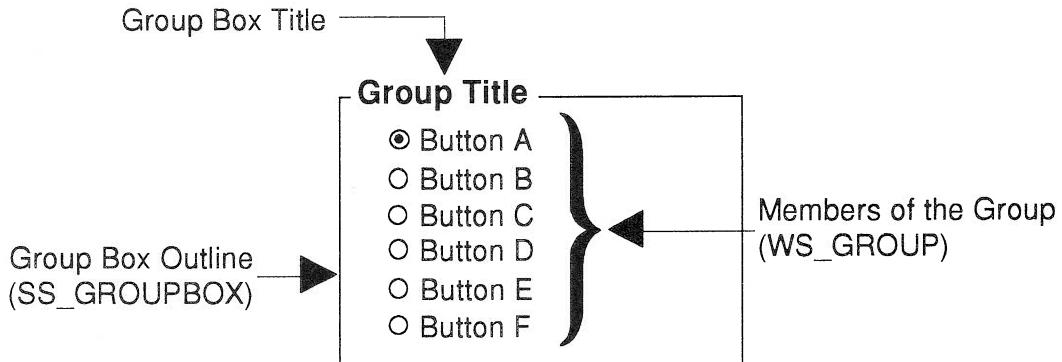


Figure 13-2 Elements of a group box

The difference between check boxes and radio buttons requires a significantly different programming approach to each control. Only one radio button may be selected at any point, so the dialog procedure needs to remember only the last button selected. Any number of check boxes may be selected, and the selecting of one does not imply the clearing of the others. As a result, the dialog procedure must maintain the state of each box.

Entry Fields

Entry fields are controls that allow the entry of alphabetic, numeric or mixed alphanumeric data. A single line entry field consists of a rectangular window into which the user can type a single line of text. A multiline entry field, its name implies, allows multiple lines of text to be entered into the control. When the entry field has the focus, the cursor is displayed within the field as a vertical bar, called the Text Cursor, and the Insert and Delete keys are enabled for text entry.

Single Line Entry Fields

Single line entry fields have styles that affect the way data is displayed within the rectangle. The style of an entry field is specified during dialog creation. An entry field can have the following styles.

ENTRY FIELD STYLES

ES-LEFT	The text is left justified (default).
ES-CENTER	The text is center justified.
ES-RIGHT	The text is right justified.
ES-AUTOSCROLL	Moving off the end of the line automatically scrolls 1/3 of the window's width.
ES-MARGIN	Text is drawn in the rectangle within a margin.
ES-AUTOTAB	When the field is filled, an automatic tab to the next field is generated.
ES-READONLY	No text may be entered into the box.
ES-COMMAND	The entry field has the style of a command entry field.
ES-UNREADABLE	Entered text is not displayed.
ES-ANY	The text is a mixture of SBCS and DBCS characters.
ES-SBCS	The text is single-byte characters only.
ES-DBCS	The text is double-byte characters only.
ES-MIXED	The text is a mix of DBCS and SBCS which may be converted into EBCDIC DBCS at a later time.

There is no corresponding auto-entry field as there are for radio buttons and check boxes; control over the operation and contents of the field is the responsibility of the dialog procedure. The procedure is notified via a WM-CONTROL message when the field gains or loses focus, has its contents changed or is scrolled. The procedure can clear, copy, cut, paste, query, set limits upon, shift or collect the data from an entry field via control messages. As with other dialog controls, notification and control is accomplished through messages sent between the dialog procedure and the dialog controls.

Grouping User Controls

Radio buttons and check boxes introduce the concept of grouping dialog controls, tying together individual controls so they operate as a single unified entity. Grouping is important because the operation of buttons and boxes, and the flexibility the user has in making a selection, is affected in a large part by the grouping of the controls. There are two window styles that aid in the grouping of controls, SS_GROUPBOX and WS_GROUP. The difference between the two styles is important to the correct operation of grouped controls.

SS_GROUPBOX defines a static rectangular box with a title, drawn on the dialog. It may contain buttons, boxes, static text, entry fields, or any controls that need the visual appearance of being related under a common title. The SS_GROUPBOX control supplies

this visual grouping, but does not exercise any influence over the controls within its boundaries. Placing radio buttons within an SS_GROUPBOX, for example, has no bearing on the relationship of the buttons and does not cause them to operate as a group. Figure 13-3 shows the SS_GROUPBOX controls *WinEditGpBox1* and *WinEditGpBox2* defined for the Window Edit dialog. While the *WinEditGpBox1* group box gives the radio buttons the appearance of a related group, it does not actually group them.

Actual control over how buttons, boxes and entry fields are grouped is determined by the window style of WS_GROUP. Any control that is given the style of WS_GROUP begins a new control group. All controls that follow, up to the next control with the style of WS_GROUP are considered to be within a single group. Any control can be defined with the window style of WS_GROUP and when WS_GROUP controls are adjacent, as with the Window Edit's *OK* and *Cancel* push buttons, single item control groups are defined.

The following example taken from the sample program shows how to group related controls using both the SS_GROUPBOX and WS_GROUP controls. The control *Window Colors* is a static control with the style of SS_GROUPBOX. This means *Window Colors* is a rectangular box with the title of *Window Colors* and the size and location defined. The grouping of the radio buttons is accomplished by giving the first radio button control (*ID-BN-Blue*) the window style of WS_GROUP. This marks the beginning of a group of controls. All controls following the *ID-BN-Blue* radio button up to the next control with the window style of WS_GROUP are considered a single related group. For the sample program, this is all of the window color radio buttons.

Notice that the next control with a style of WS_GROUP is the first check box, *ID-CB-MainWindow*. This means that the group box *ID_WinEditGpBox2* is technically part of the radio button group. But, as a static control, its position within a group is unimportant.

```

CONTROL "Window Colors", ID_WinEditGpBox1,
        18, 53, 183, 89,
        WC_STATIC, SS_GROUPBOX |
        WS_VISIBLE

CONTROL "Blue", ID_BN_Blue,
        18, 53, 183, 89,
        WC_BUTTON, BS_AUTORADIOBUTTON |
        WS_VISIBLE | WS_GROUP

```

The defining of a group affects the use of the Tab key. When a group consists of single selection items such as radio buttons, the Tab key positions the cursor at the currently selected item in the group. The next tab takes the user to the next group, not the next item within the group. Movement within a single selection group is controlled by the four Arrow keys. When the group contains multiple selection items such as check boxes, the

Tab key positions the cursor at the first, or currently selected, item in the group. Each subsequent tab takes the cursor to the next item within the group that has the style of WS_TABSTOP. Tabbing from the last item in the group with the style of WS_TABSTOP takes the cursor to the next group or next control in the Resource File with the window style of WS_TABSTOP.

Creating The Window Edit Dialog Box

To support the use of these new dialog controls, this chapter adds the Window Edit dialog box to the sample program. The Window Edit dialog box contains three button selection controls - auto-radio buttons for the specification of window colors, check boxes for the specification of windows, push buttons to allow the user to signal end of the dialog - and a single line data entry control to allow entry of a Main window title.

I will not elaborate on the building of this dialog. If you are not sure about creating the Window Edit dialog template, please review Chapter 10. Figure 13-3 shows the controls that must be specified for the Window Edit dialog box.

It is important to note that for this chapter only, none of the data entered through the Window Edit dialog is saved beyond the program's termination. The saving of this data will be discussed in the next chapter along with user and system profile processing.

Updating The Resource Script File

As with the other dialogs used in the sample program, a reference to the Window Edit dialog must be added to the Resource file, so that it will be included when the Resource file is compiled and bound with the program. Adding a dialog requires adding the following four new statements to the Resource Script File.

- A #include statement for the Winedit dialog header file.
- An RCINCLUDE statement pointing to the Winedit dialog template file.
- An Accelerator Table entry supporting the Window Edit menu entry.
- A MENUITEM entry for the Window Edit menu item.
-

Remember, the #include statement is required only if the DLGINCLUDE statement is removed from the dialog template and the RCINCLUDE statement is required only if the dialog template is not included in the Resource Script file. For a complete description of adding menu items to the Resource Script file, see Chapters 6 and 7. The Resource

Script file is included in Appendix A.

Adding The MI-Wedit Command Routine

To initiate a program action, the user must select an entry from the sample program's Action-bar or one of its pull-down menus. This selection generates a WM-COMMAND message with a command ID of the selected item. To service these commands, an evaluation routine is included in the sample program's *MainWndProc* procedure. To

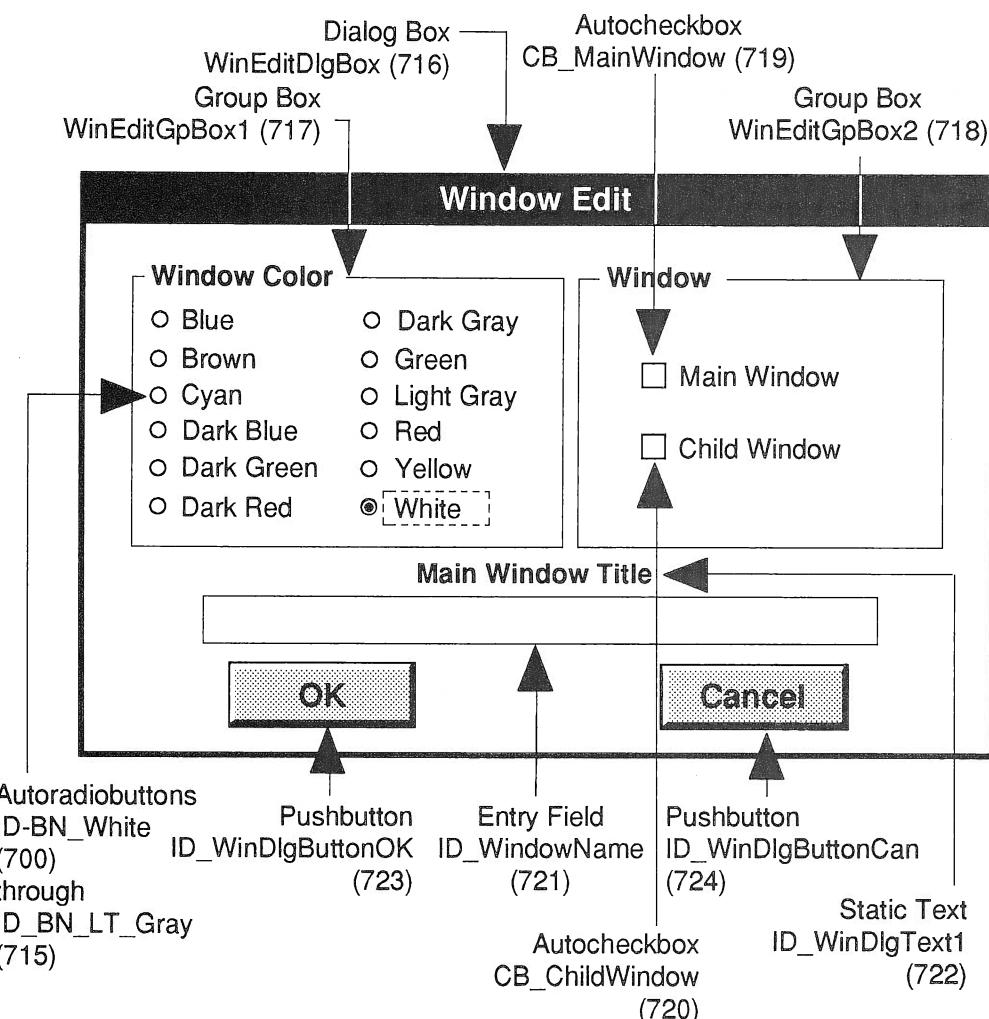


Figure 13-3 Resources for the Window Edit dialog box

support the Window Edit dialog, a *when MI-Wedit* paragraph must be added to the evaluation routine in *MainWndProc*.

The creation of the dialog box via the WinDlgBox call is detailed in Chapters 10 and 11. If you are unfamiliar with the WinDlgBox call, please review these two chapters.

Dialog Initialization Processing

When a window is created, an initialization message is sent to the window procedure, giving control to the procedure before the window is displayed. This allows the window procedure to customize the window before the user sees it. The same is true for dialogs. When a dialog is created, a WM-INITDLG message is sent to the dialog procedure, allowing the procedure to modify the dialog before it is displayed to the user. This initialization processing can include almost any dialog function. For the PM Call dialog, initialization processing included saving the CREATEPARAMS data structure pointer, loading the list box and centering the dialog box on the desktop. For the Window Edit dialog, initialization processing includes saving the CREATEPARAMS data structure pointer, selecting the initial Window Color radio button, limiting the size of the title text and centering the dialog box on the desktop.

The code required to save and set the CREATEPARAMS data structure pointer for this dialog is identical to that used for the PM Call dialog in Chapter 11. If you are unfamiliar with passing a data structure pointer to a dialog procedure, saving the pointer or establishing addressability to the data structure, please review Chapter 11.

Setting Default Dialog Controls

Whenever a number of choices is offered to the user, a default selection should be made prior to displaying the dialog. For Window Colors radio buttons, the user's last color selection will be the default color selection for the next iteration of the dialog. To set the default button, a BM-SETCHECK message specifying the control ID of the button last clicked by the user is sent to the Window Edit dialog procedure. The value sent *DWE-BN-Color* is initially defined as 700, the resource value of *ID-BN-White*, making the white radio button the default button for the first use of the Window Edit dialog. As radio buttons are selected, the control ID of the selected button is saved in *DWE-BN-Color* and used to set the radio button when the dialog box is next executed.

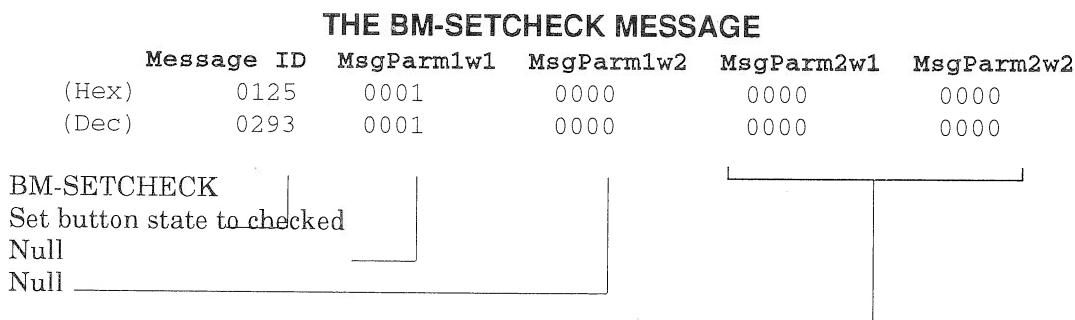
The BM-SETCHECK command is sent using the standard WinSendDlgItemMsg call. This call is coded by placing the variable holding the handle of the dialog in parameter one. As with other dialog calls, the dialog handle is the handle of the current message,

so parameter one is coded simply as *hwnd*. Parameter two is the ID of the dialog control that is to receive the message. For the sample program, this is the button ID stored in the variable *DWE-BN-Color*. Parameter three is the message ID, BM-SETCHECK. Parameters four and five are the two message parameters for the BM-SETCHECK message. For this message, message parameter one sets the check state of the control. Message parameter two is not used and should be set to null. The last parameter is the variable that is to receive the call's return code. For the BM-SETCHECK call, this return code indicates the old state of the button.

Here are the possible settings for message parameter one:

- 0 = Display the button control in the unchecked state.
- 1 = Display the button control in the checked state.
- 2 = Display a 3-state button control in the intermediate state.

Here is the format of the BM-SETCHECK message:



Here is the WinSendDlgItemMsg call used to select the initial Window Color radio button:

```

018740      Move 1 to MsgParm1w1
018750      Move 0 to MsgParm1w2 MsgParm2
018760      Call OS2API 'WinSendDlgItemMsg' using
018770          by value hwnd
018780          by value DWE-BN-Color
018790          by value BM-SETCHECK
018800          by value MsgParm1
018810          by value MsgParm2
018820          returning ReturnData

```

After selecting the radio button, the dialog procedure sets the maximum number of

characters that the user may enter for the Main window title. The string entered by the user must not exceed the length of *DWEMainTitle*, the buffer used to hold the string, minus 1 for the string's null termination character. Since the *DWEMainTitle* data structure entry is declared as 31 bytes, the maximum title length the user may enter must be restricted to 30 characters. This limit could be imposed when the title text is transferred from the dialog control into the program, but waiting until the text is transferred may cause some characters to be lost. Setting the limit into the control not only ensures no characters will be lost, but also alerts the user to the limit at the time the excess characters are entered.

Limiting entry field input is accomplished by issuing the WinSendDlgItemMsg call to send a EM-SETTEXTLIMIT message to the dialog specifying the text limit for the entry field. This call is coded exactly the same as the prior WinSendDlgItemMsg calls with the following changes: The control ID is set to *ID-WindowName*, the command is set to EM-SETTEXTLIMIT, and message parameter one is set to the maximum number of characters that may be entered.

Here is the message used to limit the length of the Main window title:

THE EM-SETTEXTLIMIT MESSAGE					
Message	ID	MsgParm1w1	MsgParm1w2	MsgParm2w1	MsgParm2w2
(Hex)	0143	001E	0000	0000	0000
(Dec)	0323	0030	0000	0000	0000
EM-SETTEXTLIMIT					
Text limit					
Null					
Null					

Here is the WinSendDlgItemMsg call used in the sample program to set the maximum text length allowed in the entry field:

```

018860      Compute MsgParm1w1 = DWELength - 15
018870      Move 0 to MsgParm1w2 MsgParm2
018880      Call OS2API 'WinSendDlgItemMsg' using
018890          by value hwnd
018900          by value ID-WindowName
018910          by value EM-SETTEXTLIMIT
018920          by value MsgParm1
018930          by value MsgParm2
018940          returning ReturnData

```

After setting the limits on text input, one additional call is made to set the change

indicator for the entry field. The EM-QUERYCHANGED call, used to determine if the contents of the ID-WindowName entry field have been changed, returns the changed state since the last query. Sending the EM-SETTEXTLIMIT message to this control caused its changed state to be set to true. So to ensure that the EM-QUERYCHANGED messages issued during the dialog accurately reflect user activity, set the initial control status to unchanged by issuing an initial EM-QUERYCHANGED message to *ID-WindowName* and ignoring the returned value.

Finally, the initialization routine centers the dialog box on the desktop using the standard routine that queries for the size of the desktop, queries for the size of the dialog box, then sets the dialog box's position using the values returned by these two queries. For a complete discussion of the calls used in this routine, see Chapter 10.

Coding The Window Edit Dialog Procedure

While the dialog is active, the Window Edit dialog procedure only manages the state of the radio buttons and window check boxes. The user can change the state of these buttons and boxes as often as desired, so the dialog procedure tracks the current state. When the dialog is closed, the current state of the controls becomes the selected state. Since activity for both of these controls is reported via the same WM-CONTROL message, the dialog procedure intercepts only the WM-CONTROL message to manage the state of these controls, plus the WM-COMMAND message to determine when the dialog is to be ended.

Processing Check Box Data

Processing of check boxes involves intercepting the BN-CLICKED messages that apply to the check boxes, querying for the state of the selected box, and recording the correct state for use by the Main window Procedure. Here is the logic used to process the Window Edit check boxes in the sample program:

```
When WM-CONTROL
    Establish data structure addressability
    If BN-CLICKED message
        If check box message
            Compute last color value selected
            Query for current box state
```

```

If Main window check box
    Set color as Main window color
Else
    Set color as Child windows color
End-if
End-if
End-if

```

Check box processing does not require sophisticated coding, simply keeping an accurate record of the current state of each box. By querying for the current state of the box with every BN-CLICKED message, the procedure is assured that the recorded state of the box matches the displayed content of the box (checked or blank).

Normally, the dialog has no requirement to understand the meaning of a check box, only accurately record the state of each box. There are, of course, variations in check box processing, depending upon the dialog. Selecting a check box, for example, may require the resetting or reloading of other dialog controls. In this case, the state of the selected control must be acted upon as well as recorded. But regardless of the dialog's logic, three basic functions are required by every dialog when processing check boxes. They are:

- 1) Determine which box has been selected.
- 2) Query for the current state of the box.
- 3) Record the state for later use.

Once the sample program determines that a check box was selected, it issues the BM-QUERYCHECK message to determine the current state of the box. The call's return code indicates the current state of the box. Here is the content of the BM-QUERYCHECK message used in the sample program's Window Edit dialog procedure:

THE BM-QUERYCHECK MESSAGE

Message ID	MsgParm1w1	MsgParm1w2	MsgParm2w1	MsgParm2w2
(Hex) 0124	0000	0000	0000	0000
(Dec) 0292	0000	0000	0000	0000

BM-QUERYCHECK
Unused _____

With the correct state of the check box, the dialog procedure determines which box was selected by the user and sets the correct state. If the query returned true (checked), the

dialog procedure sets the value of the last color selected into *DWEMWindowColor*, if the Main window was selected, or into *DWECCWindowColor* if child windows was selected. If the returned state of the window check box was false (unchecked), the related window color save area is cleared to 0.

Processing Radio Button Data

Radio buttons are processed in a similar manner to check boxes. The dialog determines which button was selected and saves the necessary information related to that button. When using auto-radio buttons, processing is simpler as the procedure need not worry about the prior status of any button. Since only one button may be selected at any point, recording the ID of the button selected ensures that the state of all the other buttons within the group is cleared.

Processing of auto-radio buttons involves intercepting the BN-CLICKED messages that apply to radio buttons. The procedure then records the ID of the button selected and arbitrarily sets the text color to black. The procedure then evaluates the window color selected to see if it is one of the six darker shades that requires white text rather than black text. If any of these six colors was selected, the color of the text is changed to white. Here is the logic used to process the Window Color radio buttons:

```
If button selected was a radio button
  Record resource ID of button selected
  Set text color to black
    Evaluate button ID for dark colors
      when dark color selected, set text color to white
    End-evaluate
  End-if
```

The check box and radio button messages are the only messages of interest while the dialog is active. All other messages are returned to PM via the WinDefDlgProc call.

Processing Entry Field Text

There are three ways that entry field data can be processed. Each of these alternatives uses the same calls, but depending upon the amount of editing required, the calls are implemented at different times within the dialog routine.

For extremely fine editing, each character can be obtained and examined as it is entered by the user. For every change to the entry field made by the user, the dialog routine receives an EN-CHANGE message. Upon receipt of this message, the dialog procedure can query the Entry Field for the changed characters and edit them. Be aware, however, that the changed characters are not necessarily at the end of the string. This type of processing will allow the dialog routine to edit at the character level and immediately signal the user when an error is detected. While this type of processing improves the dialog's edit responsiveness to user input, it will significantly increase the processing workload of the dialog procedure and thus the CPU.

For less immediate editing, you can obtain the text whenever the entry field loses the focus, indicating that the user has completed entering or changing the contents of the field. Whenever the entry field loses focus, the dialog procedure receives an EN-KILLFOCUS message. When this message is received, the dialog procedure can query the change status of the entry field. If there has been a change, then the procedure retrieves and edits the text. This routine delays editing until the user has completed entering or changing the data, but allows retrieval of the text as a single string and reduces the processing workload of the dialog procedure. This is the most common method of capturing and editing entry field text.

You must be very careful when using the second method to validate entry field data. Receipt of the EN-KILLFOCUS message means that the user has shifted the focus to another control window. The dialog procedure is now faced with editing data for a field that no longer has the focus or the cursor. If the dialog procedure wishes to alter or highlight the data in the entry field that it is editing, it must first move the focus and the cursor back to the entry field being edited. But, movement between dialog controls is the province of the user, and the dialog procedure can't manipulate the focus and cursor without the user's knowledge and consent. The dialog procedure must be absolutely sure that the user has been made aware of any focus change prior to the movement by the procedure. You can imagine how confusing it would be to a user typing data into one field to suddenly find the focus and cursor moved to the prior field, with the text for the current control being entered into the prior control.

Where no editing is required, as in the sample program, or when editing of the data is very complex and requires a significant amount of time, the entry field can be queried when the dialog is complete. If a change has occurred the data can be edited or saved for use by the main program. When text retrieval is delayed to the end of the dialog, as is done with this method, the user is not informed of any errors until long after the data is entered. Normally, this delay is not acceptable, but there are circumstances when this method may be desirable. Where a long validation is required, such as database look-ups, this method is used together with a delay in dismissing the dialog. Any errors encountered are then reported to the user through a pop-up message box. This method certainly produces the lowest processing workload for the dialog procedure, as all changes to the entry field can be ignored until the dialog is completed.

Which method of editing entry field data to use depends upon the requirements of the program and the function of the dialog. But you should always try to be as responsive to the user as possible, and being responsive when editing data means notifying the user of an error as close to the creation of the error as possible.

In the sample program, when the user signals the end of the dialog, the dialog routine queries the change status of the ID-WindowName Entry Field to determine if the user has entered or changed data in the field. This call is coded exactly like the other WinSendDlgItemMsg calls, with the following exceptions: The control ID is set to *ID-WindowName*, the command is set to EM-QUERYCHANGED, and the two message parameters are set to null. Here is the EM-QUERYCHANGED call used in the Window Edit dialog procedure:

THE EM-QUERYCHANGED MESSAGE					
Message ID	MsgParm1w1	MsgParm1w2	MsgParm2w1	MsgParm2w2	
(Hex) 0140	0000	0000	0000	0000	
(Dec) 0320	0000	0000	0000	0000	
EM-QUERYCHANGED					
Unused _____					

At the completion of the call, the returned data will be true if the text has been changed, or false if no change has occurred. It is important to note that the status of the change flag reflects activity only since the last query. The change flag is set to false with every EM-QUERYCHANGED message. If the change status is queried more than once, the dialog procedure will need to determine the overall change status of the field.

To retrieve text from an entry field, the handle of the entry field must first be obtained by issuing the WinWindowFromID call. The WinWindowFromID call returns the handle of the child window identified by the resource ID passed with the call. To code this call, place the variable holding the dialog procedure handle, the current message handle, in parameter one. Parameter two contains the resource ID of the control window whose handle is required. Parameter three is a long variable that will contain the handle of the window when the call completes. Here is the WinWindowFromID call used in the Window Edit dialog procedure:

```

019440           If ReturnTrue
                  Call OS2API 'WinWindowFromID' using
019480                 by value hwnd
019490                 by value ID-WindowName
019500
019510                 returning LongWork

```

Using the returned handle, the text is retrieved by querying for the text of the window. To do this, first set a limit on the number of characters that can be transferred to the size of the program's buffer, then issue the WinQueryWindowText call. Even though the length of the field was limited at the start of the dialog, this call requires the the limit be passed along with this call.

To code the WinQueryWindowText call, place the variable holding the handle of the entry field control window in parameter one. This is the window handle returned in the prior WinWindowFromID call. Place the maximum length of the text that may be returned in parameter two. Parameter three is the variable holding the pointer to the buffer that is to receive the returned text, and the last parameter is the variable that will contain the length of the returned text. Here is the WinQueryWindowText call used to retrieve the user entered Main window title text:

```

019530      Compute ShortWork = DWELength - 15
019540      Call OS2API 'WinQueryWindowText' using
019550          by value      LongWork
019560          by value      ShortWork
019570          by reference DWEMainTitle
019580          returning     ShortWork

```

With the state of the check boxes and radio buttons stored and the Main window title saved, the dialog procedure is ended by dismissing the dialog. For a complete description of how to code the WinDismissDlg call, see Chapters 10 and 11.

Processing The Dialog's Returned Data

As with all dialogs, when control is returned to the window procedure that created the dialog, a test must be made to determine if the dialog completed successfully or was canceled by the user. This test will differ, depending upon the type of controls used by the dialog and how data is returned from the dialog. However, the test for successful completion of the dialog must be the first function performed upon completion of the WinDlgBox call. Failure to perform this test first may allow data entered into the canceled dialog to update program data.

In the sample program, the ID of the push button selected by the user is returned as the WinDlgBox call's return code. So, immediately following the WinDlgBox call, a test is made to determine which push button was selected by the user. If the *Cancel* push button was selected, then the *MI-Wedit* routine is exited and any data entered by the user ignored. If the *Cancel* push button was not selected, processing of the returned data continues.

A window's Title bar is part of the window's frame window. So, if a Main window title was entered by the user, this text string must be passed to the Main window's Frame window for display using the WinSetWindowText call. This call is coded by placing the variable holding the handle of the target frame window in parameter one. The frame window handle was returned as the return code from the WinCreateStdWindow call. For the sample program, the Main window's Frame window handle is stored in the variable *hwndFrame*. Parameter two is a pointer to the buffer holding the null-terminated title text string. Parameter three is the variable that is to receive the call's return code. Here is the WinSetWindowText used to add the title to the Main window:

```
011930      If WEMainTitle is not equal to spaces
011940          Call OS2API 'WinSetWindowText' using
011950              by value      hwndFrame
011960              by reference WEMainTitle
011970              returning    ReturnData

012080      Move spaces to WEMainTitle
012090  End-if
```

Following the setting of the Main window's title, the text is cleared from *WEMainTitle* to prevent resetting the title after the next iteration of the dialog, if no change was made to the title.

With the title set, the window edit procedure determines if a window color was selected by the user. A test for *WEMWindowColor* greater than 0 and *WECWindowColor* greater than 0 determines if the user selected a color for either the Main window or the child windows. No setting of the actual values to use is required, as the correct color value was calculated within the dialog procedure and the window painting routine uses that color, stored in the Window Edit data structure.

If a color was selected, the window, if currently displayed, needs to be repainted immediately using the requested color. Repainting is accomplished whenever the WM-PAINT message is received by a window procedure, and a WM-PAINT message will be generated by PM whenever it is determined that a window, or part of a window, is no longer displayed correctly. This invalid condition can be forced by issuing the WinInvalidateRegion call. So, if the user has selected a new color for the Main window's Client window, the window edit procedure issues the WinInvalidateRegion call, using the Main window's Client window handle to cause the entire Main window's Client window to be repainted.

To code the WinInvalidateRegion call, place the handle of the window containing the region to be invalidated in parameter one. For the sample program, this is the handle of the Main window's Client window, which is also the handle of the current message.

So, the first parameter is simply coded as *hwnd*. Parameter two is used to specify the subregion of the window that is to be added to the window's update region. This subregion must be established prior to this call and referenced via the subregion's handle. Parameter two contains an entry only when part of the window is to be invalidated. If the entire window is to be invalidated, as it is in the sample program, then parameter two is coded as a Null. Parameter three specifies how children of the window are to be processed. If all children of the window are to be added to the region to be invalidated, code parameter three as true. If all children of the window are to be excluded from the invalidated region, code parameter three as false. Regardless of what is coded, if the parent window was created with a style of WS-CLIPCHILDREN, then all child windows will be included within the invalidated region. For the sample program, parameter three is coded as false to avoid repainting any child windows. The last parameter is the standard return code save area. Here is the WinInvalidateRegion call used in the sample program:

```

012270      If WEMWindowColor is not equal to 0

012290          Call OS2API 'WinInvalidateRegion' using
012300              by value hwnd
012310              by value LongNull
012320              by value SetFalse
012330              returning ReturnData

```

Following the setting of the Main window color, a similar routine exists for the child windows. The routine first checks *WECWindowColor* for a color value and, if a value was returned, each of the client windows of all active child windows must be repainted.

At this point, however, a problem presents itself. There is no indication within the program of which, if any, child windows are active. Indeed, up until this point there has been no need to know which windows are active because the procedure supporting each window is automatically dispatched by PM as messages arrive.

Window Enumeration

To allow a procedure to determine which windows are open at any time, PM provides a function called window enumeration. Window enumeration identifies all the immediate child windows of a specified window, including any invisible or minimized windows. It is important to note that window enumeration identifies only the immediate child windows, usually the frame window of each child window. The window enumeration process is composed of three Presentation Manager calls, WinBeginEnumWindows, WinGetNextWindow and WinEndEnumWindows.

Issuing the WinBeginEnumWindows call causes PM to construct a list of all the active windows that are the immediate children of the window specified by the handle passed in the call. This list is constructed to avoid the problem of windows being opened and closed during the enumeration process.

The WinGetNextWindow call returns the handle of each window on the enumeration list. To enumerate all the immediate child windows, continue to issue the WinGetNextWindow call until PM returns Null as the window handle, indicating the end of the list.

The WinEndEnumWindows call completes the enumeration process by notifying PM that the enumeration list may be destroyed.

For many uses, such as tiling or cascading windows, the enumeration process works well using this BEGIN, GETNEXT, END process. However, where the need exists to enumerate windows at a lower level than the immediate child window, the enumeration process must be nested to achieve the correct results. The determination of whether nesting is required lies with the handles available to the program. If the program has access to a handle that is one level above the desired windows, then a simple enumeration will return the desired handles. However, where a program does not have access to the parent's window handle, or where the number and depth of windows is unknown, then enumeration routines must be nested to retrieve all the required window handles. Nested window enumeration takes the following logical form.

Figures 13-4 and 13-5 show the process of window enumeration and how window handles may be obtained using a window enumeration loop. Enumerating the immediate

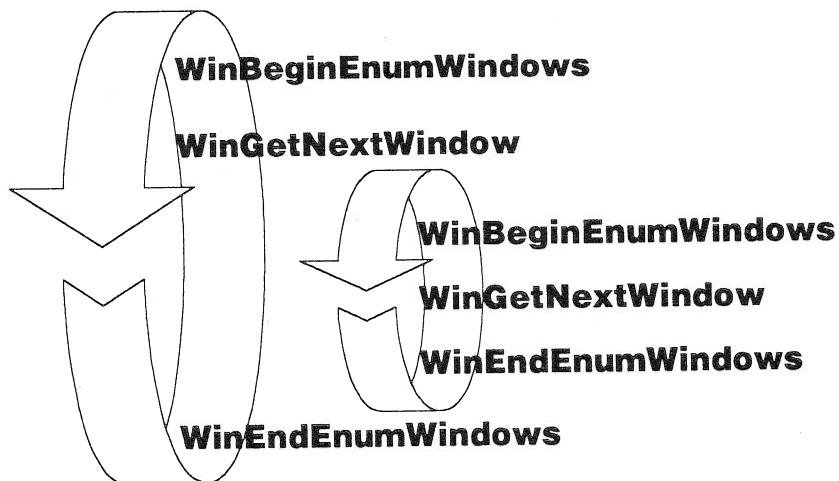


Figure 13-4 Nested window enumeration loops

children of the Main window's Client window returns the handle of the frame window of the first child window (*hwndSource*) with the first call. Succeeding WinGetNextWindow calls will return the handles of succeeding immediate child window frame windows until all immediate child window frame windows have been enumerated. The end of the enumeration list is indicated when a Null is returned as the window handle. In Figure 13-5, the enumeration of the immediate child windows is shown in the upper horizontal box labeled *Main Enumeration*.

To reach the next lower level of window, each frame window handle returned by the top level WinGetNextWindow call is used as the argument to start another enumeration process. This second enumeration process returns the immediate children of each frame window until this list is exhausted and a Null is returned. For the sample program, the Source Code window's Title-bar, the Action-bar and Client window are returned. In Figure 13-5, the subwindow enumeration is shown in the lower horizontal box labeled *Subwindow Enumeration Loops*. It is possible with nested enumeration to identify every window within your program knowing only the handle of the desktop.

The second level enumeration loop returns the handles of multiple windows for each child window. The process of positively identifying which handle represents client windows requires querying the windows represented by the returned handles. When searching for a specific child window, the query for the type of window represented by the returned handle is essential. There is no indication given by the WinGetNextWindow call as to the type of window the returned handle represents. Fortunately, this is not a requirement for the sample program; window invalidation does not cause a change in the window's appearance, but only causes the window to be redrawn using the new color. Thus, the sample program simply invalidates every window handle returned in the second level enumeration loop. The Title-bar and Action-Bar will be redrawn by their default procedures and will not be visible to the user. The Main window's Client window is also redrawn, but using the new color saved in *WECWindowColor*. It is important to point out that if many windows were open, or if complex windows were involved, then the program should determine the exact type of window represented by the returned handle and invalidate only those windows that need redrawing, saving time and processor cycles.

Coding The WinBeginEnumWindows Call

To code the WinBeginEnumWindows call, place the variable holding the handle of the window whose immediate children are to be enumerated in parameter one. For the sample program, the enumeration routine starts with the handle of the Main window's Client window. Parameter two is the variable used to store the handle of the generated enumeration list. Here is the WinBeginEnumWindows call used in the sample program:

```

012670      Call OS2API 'WinBeginEnumWindows' using
012680          by value hwnd
012690          returning hwndChildWindows

```

Coding The WinGetNextWindow Call

To code the WinGetNextWindow call, code parameter one with the variable holding the handle of the enumeration list returned by the WinBeginEnumWindows call. Parameter two is the variable used to store the handle of the next window returned from the enumeration list. Here is the sample program's top-level WinGetNextWindow call :

```

012760      Call OS2API 'WinGetNextWindow' using
012770          by value hwndChildWindows
012780          returning hwndChildQuery

```

Coding The WinEndEnumWindows Call

The WinEndEnumWindows call requires only the variable holding the handle of the completed enumeration list as parameter one and the standard return code variable as parameter two. Here is the WinEndEnumWindows call used to terminate the top-level enumeration process in the sample program:

```

013020      Call OS2API 'WinEndEnumWindows' using
013030          by value hwndChildWindows
013040          returning ReturnData

```

Changing The Color Of Window Text

Giving control over the window colors to the user requires that the sample program take care to use a text color that contrasts with the window's background color. The proper text color, black or white, was determined during the Window Edit dialog and stored in the *WETextColor* variable of the *WEEdit* data structure. To enable drawing the text using the correct contrasting color, parameter two of the GpiSetColor call has been changed to reference the *WETextColor* variable. This ensures that anything drawn into the window will use the correct contrasting color.

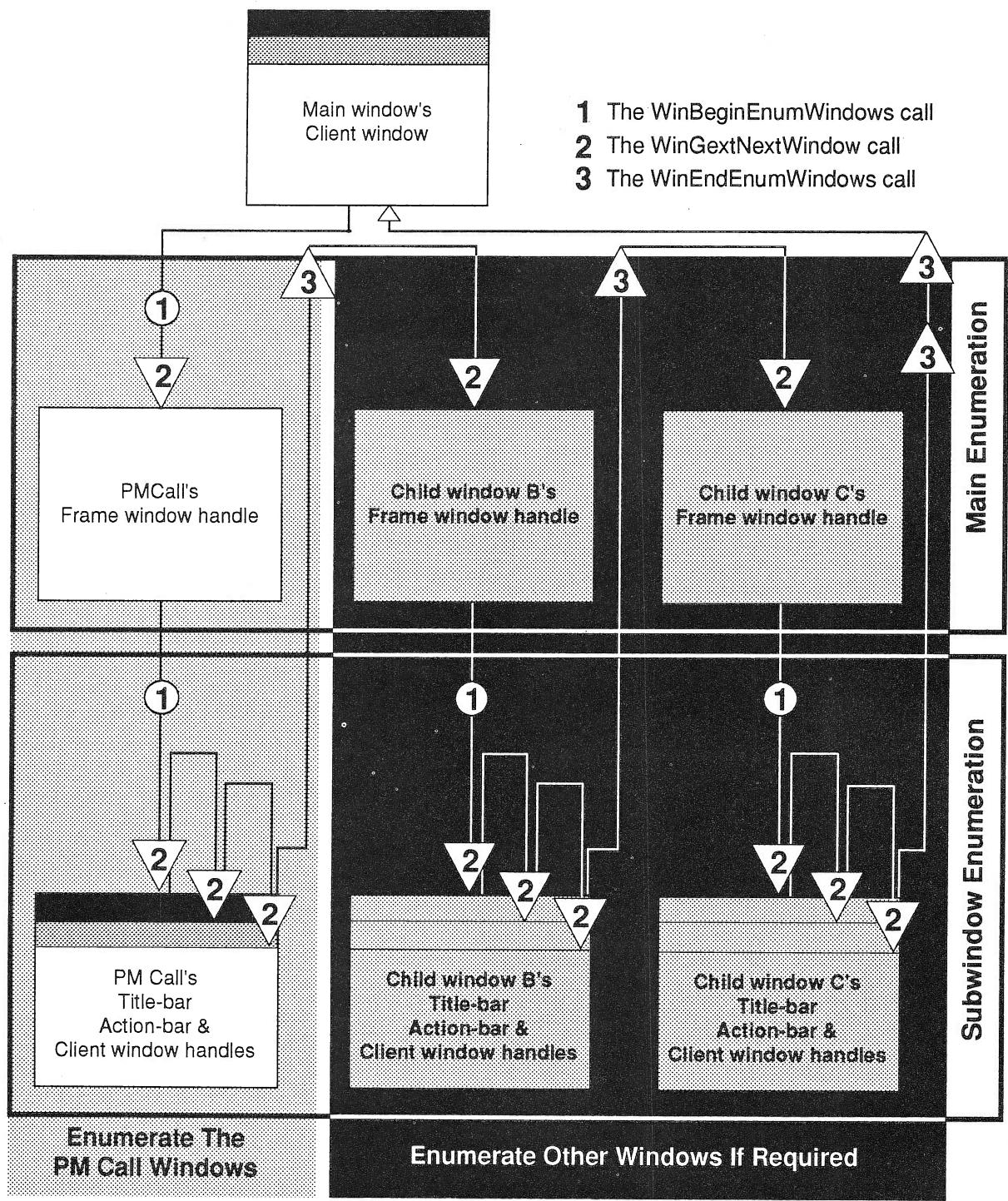


Figure 13-5 The process of enumerating child windows

Chapter 14

System And Application Profiles

One of the most confusing aspects of OS/2 is the role played by the user initialization file, OS2.INI and the system initialization file, OS2SYS.INI. While users are generally aware that these files exist, and may even have received error messages indicating that one or both of these files had been corrupted, most users and many developers do not fully understand the structure and use of these initialization files.

Initialization files, more commonly referred to as profiles, are external storage where OS/2, the Presentation Manager or an application can store software and hardware customization information needed to preserve and recreate the environment as defined by the user. Initialization files - from this point on, I will refer to them as profiles - are somewhat similar to resource files in that they hold ASCII data externally to the program using it. But there several differences between profiles and resources.

- Profiles are treated like files. They must be opened before use and closed after use.
- All active profiles are considered a PM resource and data must be read or written through the Presentation Manager.
- Profiles store only two types of data, ASCII string and ASCII binary data.
- All data within a profile is stored in a two-tiered index format.

The definition of what constitutes customization information is really up to the program using the profile. Although the two system profiles contain specific types of information, almost anything that can be stored in one of the two ASCII formats can be placed into any profile.

There are two categories of profiles, those created and owned by the OS/2 Kernel, OS/2 add-on modules and the Presentation Manager and those created and owned by individual applications. The distinction between these two categories lies in the data they hold and the role each plays for the owning program, and not in their structure. The structure and organization of each category of profile is identical. The data is stored in identical formats and organization, and the data is accessed using the same PM API calls.

The System Profiles

There are two system profiles with every OS/2 system. Depending upon what parts of OS/2 your machine is running there may be more, but the default user profile and the system profile can be found on every OS/2 machine.

The default user profile, OS2.INI, contains customization information required by PM to correctly start and maintain the system as defined by the user. The OS2.INI profile contains information about the program groups, the programs contained within each group, file type association, installed fonts, country dependent information and the current window colors. The system profile, OS2SYS.INI, contains information about the installed hardware and the structures defined by the user to support the hardware. A majority of the information within the OS2SYS.INI file concerns the Print Manager and its related print queues, print drivers, printer properties and ports. The information held in these two profiles should be of interest to every program, making the querying of these profiles a necessary function for programs that wish to automate much of the user's decision making.

PM API calls are available to allow a program to maintain its own information in either of these files. While it is strongly recommended that programs do not add information to these system profiles, it is technically possible to add and maintain application customization information within these default user profiles. There are many reasons why applications should maintain their own profile as opposed to placing data into one of the system profiles. Here are some things that may discourage you from using the system profiles.

- The two system profiles are considered OS/2 resources and, as such, their contents are subject to change from one release of OS2 to another. There was one major change already when the single profile used in early releases of OS/2 was divided into the current two system profiles.
- As an OS/2 resource, system profiles are open from system startup to shutdown, making it impossible to copy or back up these files while PM is running.

- Information stored in the system profiles may be lost when a new version of OS2 is installed, or if the profile becomes corrupted and requires rebuilding.
- System profiles offer read-write access to any program running in the system and are more prone to damage from incorrect access. Failures in the operating system itself often lead to damage of the profiles.

The Structure Of Profiles

Profiles store data as null-terminated ASCII strings, or ASCII binary data, organized in a three-tiered, hierarchical structure composed of two indexes and a customization string. The first level index is called the Application Name, the second level index is referred to as the Key name, or key, and the third level is the actual customization data. Here is an example of how customization data is stored within a profile:

Application Name	Key	Customization String
PROGRAM-ABC	COLORS	"1,2,3,4,5"
PROGRAM-ABC	TITLE	"This is Program ABC"
PROGRAM-ABC	BITMAP	"C:\PROGABC\BITMAP.BMP"
PROGRAM-ABC	SWPSTRUCT	"25,20,100,75"

Application name strings identify the source of the entry and define the major entry groups. Application name strings have no fixed structure or content and may be any string that is not currently used as an application name within the profile. While the choice of an application name is arbitrary, application names usually identify the source of the entry (*PROGRAM-ABC*) or its primary use (*PM_SPOOLER*). Here are some examples of application names, taken from the user profile, OS2.INI.

```
PM_ASSOC_FILTER
PM_ASSOC_TYPE
PM_Colors
PM_ControlPanel
PM_Fonts
PM_INFO
PM_National
```

Key names control access to individual customization strings when they are stored under a common application name. But, in addition to indexing the strings, the keys help to identify the function of each string, with names like *ActiveBorder*, *Menu*, *TitleText* and

WindowText. Keys, like application names, have no fixed structure or content and may be any string not currently used as a key within the profile.

Customization strings contain the actual program or system information. The content of these strings depends upon the application that created them. For example, under the application name *PM_Fonts* and the key name of the individual font faces, the customization string points to the DASD storage location of the font using the fully qualified font file name. Under the application name *PM_National* and the key name of the type of punctuation character, the customization string contains the actual punctuation character or characters. See Figure 14-1.

Application names and keys are absolutely case sensitive, and care must be taken to ensure the correct case when creating and searching profiles because a single character in an incorrect case will result in a *not found* condition. Most application and key names are in upper case, but this is not a requirement. They may be all upper case, mixed case or all lower case. The system profiles contain both upper-case and mixed-case application names. There are no rules and in some cases no logic about when to use one case or the other. The important fact to remember is that whatever case you use, index names are case sensitive. The following application names all refer to the string program-abc, but each is a unique application name.

PROGRAM-ABC Program-ABC Program-abc PROGRAM-abc

Figures 14-1 contains a list of all application names and keys used in the OS/2 Release 1.3 OS2SYS.INI file before any user modifications. Figure 14-2 contains similar information for the OS2.INI file.

The Default OS2SYS.INI File (Version 1.3)

Application Name	Key Name	String Content
PM_INFO	Version	1.3
PM_SPOOLER	SPOOL	1;
PM_SPOOLER	DIR	C:\SPOOL;
PM_SPOOLER_QP	PPRINT	C:\SPOOL\DLL\PPRINT.QPR;
PM_SPOOLER_PORT	LPT1	;
PM_SPOOLER_PORT	LPT2	;
PM_SPOOLER_PORT	LPT3	;
PM_SPOOLER_PORT	COM1	1200;1;7;1;0;
PM_SPOOLER_PORT	COM2	1200;1;7;1;0;
PM_SPOOLER_PORT	COM3	1200;1;7;1;0;

Figure 14-1 Contents of the default System Profile

The Default OS2.INI File (Version 1.3)

Application Name	Key Name	String Content
PM_CONTROLPANEL	BorderWidth	4
PM_ASSOC_TYPE	Plain Text	C:\OS2\E.EXE
PM_ASSOC_TYPE	OS/2 Command File	C:\OS2\E.EXE
PM_ASSOC_TYPE	DOS Command File	C:\OS2\E.EXE
PM_ASSOC_TYPE	Executable	
PM_ASSOC_TYPE	Metafile	
PM_ASSOC_TYPE	Bitmap	
PM_ASSOC_TYPE	Icon	
PM_ASSOC_TYPE	Binary Data	
PM_ASSOC_TYPE	Dynamic Link Library	
PM_ASSOC_TYPE	C Code	
PM_ASSOC_TYPE	Pascal Code	
PM_ASSOC_TYPE	BASIC Code	
PM_ASSOC_TYPE	COBOL Code	
PM_ASSOC_TYPE	FORTRAN Code	
PM_ASSOC_TYPE	Assembler Code	
PM_ASSOC_TYPE	Library	
PM_ASSOC_TYPE	Resource File	
PM_ASSOC_FILTER	*.BAT	C:\OS2\E.EXE
PM_ASSOC_FILTER	*.CMD	C:\OS2\E.EXE
PM_ASSOC_FILTER	*.TXT	C:\OS2\E.EXE
PM_ASSOC_FILTER	*.DOC	C:\OS2\E.EXE
PM_Font_Drivers	PMATM	C:\OS2\DLL\PMATM.DLL
PM_FONTS	COURIER	C:\OS2\DLL\COURIER.FON
PM_FONTS	HELV	C:\OS2\DLL\HELV.FON
PM_FONTS	TIMES	C:\OS2\DLL\TIMES.FON
PM_FONTS	SYSMONO	C:\OS2\DLL\SYSMONO.FON
PM_FONTS	HELVETIC.PSF	C:\OS2\DLL\HELVETIC.PSF
PM_FONTS	COURIER.PSF	C:\OS2\DLL\COURIER.PSF
PM_FONTS	TIMESNRM.PSF	C:\OS2\DLL\TIMESNRM.PSF
PM_INFO	Version	1.3
SYS_DLLS	LoadOneTime	REXXINIT
PM_National	iCountry	1
PM_National	iDate	0
PM_National	iCurrency	0
PM_National	iDigits	2
PM_National	iTime	0
PM_National	iLzero	0
PM_National	s1159	AM

Figure 14-2 Contents of the default User Profile - Part A

The Default OS2.INI File (Version 1.3)

Application Name	Key Name	String Content
PM_National	s2359	PM
PM_National	sCurrency	\$
PM_National	sThousand	,
PM_National	sDecimal	.
PM_National	sDate	-
PM_National	sTime	:
PM_National	sList	,
PM_Colors	Display	8514
PM_Colors	ActiveBorder	255 255 0
PM_Default_Colors	ActiveBorder	255 255 0
PM_Colors	ActiveTitle	0 64 128
PM_Default_Colors	ActiveTitle	0 64 128
PM_Colors	ActiveTitleText	255 255 255
PM_Default_Colors	ActiveTitleText	255 255 255
PM_Colors	AppWorkspace	255 251 226
PM_Default_Colors	AppWorkspace	255 251 226
PM_Colors	Background	204 204 204
PM_Default_Colors	Background	204 204 204
PM_Colors	ButtonDark	128 128 128
PM_Default_Colors	ButtonDark	128 128 128
PM_Colors	ButtonDefault	0 0 0
PM_Default_Colors	ButtonDefault	0 0 0
PM_Colors	ButtonLight	255 255 255
PM_Default_Colors	ButtonLight	255 255 255
PM_Colors	ButtonMiddle	204 204 204
PM_Default_Colors	ButtonMiddle	204 204 204
PM_Colors	DialogBackground	255 255 255
PM_Default_Colors	DialogBackground	255 255 255
PM_Colors	FKAbackground	255 255 255
PM_Default_Colors	FKAbackground	255 255 255
PM_Colors	FKAtext	0 0 0
PM_Default_Colors	FKAtext	0 0 0
PM_Colors	HelpBackground	255 255 255
PM_Default_Colors	HelpBackground	255 255 255
PM_Colors	HelpHilite	0 170 170
PM_Default_Colors	HelpHilite	0 170 170
PM_Colors	HelpText	0 0 128
PM_Default_Colors	HelpText	0 0 128
PM_Colors	HiliteBackground	0 0 0

Figure 14-2 Contents of the default User Profile - Part B

The Default OS2.INI File (Version 1.3)

Application Name	Key Name	String Content
PM_Default_Colors	HiliteBackground	0 0 0
PM_Colors	HiliteForeground	255 255 255
PM_Default_Colors	HiliteForeground	255 255 255
PM_Colors	IconText	0 0 0
PM_Default_Colors	IconText	0 0 0
PM_Color	InactiveBorder	204 204 204
PM_Default_Colors	InactiveBorder	204 204 204
PM_Colors	InactiveTitle	204 204 204
PM_Default_Colors	InactiveTitle	204 204 204
PM_Colors	InactiveTitleText	0 0 0
PM_Default_Colors	InactiveTitleText	0 0 0
PM_Colors	Menu	255 255 255
PM_Default_Colors	Menu	255 255 255
PM_Colors	MenuText	0 0 0
PM_Default_Colors	MenuText	0 0 0
PM_Colors	OutputText	0 0 0
PM_Default_Colors	OutputText	0 0 0
PM_Colors	Scroolbar	224 224 224
PM_Default_Colors	Scroolbar	224 224 224
PM_Colors	Shadow	128 128 128
PM_Default_Colors	Shadow	128 128 128
PM_Colors	TitleBottom	204 204 204
PM_Default_Colors	TitleBottom	204 204 204
PM_Colors	TitleText	255 225 255
PM_Default_Colors	TitleText	255 255 255
PM_Colors	Window	255 255 255
PM_Default_Colors	Window	255 255 255
PM_Colors	WindowFrame	128 128 128
PM_Default_Colors	WindowFrame	128 128 128
PM_Colors	WindowStaticText	0 0 128
PM_Default_Colors	WindowStaticText	0 0 128
PM_Colors	WindowText	0 0 0
PM_Default_Colors	WindowText	0 0 0

Figure 14-2 Contents of the default User Profile - Part C

To retrieve a specific customization string the program must supply both the application name and the key. When only the application name is known, PM can be queried to enumerate all of the keys for a given application name, but there is no corresponding query to enumerate all the application names. So, as a minimum, the application name must be available to any program that wishes to query the related data.

Accurate data storage and retrieval requires a unique application name and key index. All customization string retrieval is done in a strict sequential manner, starting with the first entry and running through the last. It is quite common to have duplicate application names as Figures 14-1 and 14-2 show, but each application name must have a unique key. Failure to have a unique application name and key combination will result in lost data. It is possible to generate duplicate application and key combinations, and owing to the sequential searching method, the second entry of a duplicate pair, even with different customization strings, will never be found. To avoid duplicate entries, programs should always query the target profile before creating a new entry to ensure that the planned application and key combination are not already in use.

Application Profiles

Applications have the choice of placing their customization information in the default user profile, OS2.INI, or creating their own profile. Application profiles have the same two-tiered, null-terminated string index structure as do the system profiles. They are queried and updated using the same PM profile function calls and have the same null-terminated string or ASCII binary data customization string format. The only additional requirement for application profile processing is that the profile be opened before use and closed before the program terminates its PM relationship. The sample program uses an application profile, SAMPLE.INI, to hold the Window Edit customization information.

An initial version of the application profile must be built prior to its use in much the same way a resource file must be built and compiled. Using any ASCII editor, create a source profile using the format shown in the following skeleton. This file should have the file extension of .RC. Replace the application name, key and customization string shown in the skeleton with at least one of your program's application names, key names and customization strings. One entry is required to create a skeleton profile, but this initial version may include as many entries as required by the application. The ten null fields at the start of the skeleton file are required and indicate that no group entries are included; the single null entry at the end of the file delineates the last profile entry. For additional examples, review the source of the system profiles, INI.RC and INISYS.RC, found in the \OS2 subdirectory.

Here is the initial application profile created for the sample program. The profile has the name SAMPINI.RC.

After creating the skeleton resource file, a special form of the Resource Compiler called **MAKEINI.EXE** is used to compile the source file into the profile format required by PM. **MAKEINI.EXE** requires two command line parameters, the fully qualified name of the output file followed by the fully qualified name of the source input file. Here is the **MAKEINI.EXE** command used to compile the sample program's user profile:

MAKEINI	<u>d:\directory\SAMPLE.INI</u>	<u>d:\directory\ SAMPINI.RC</u>
Optional drive and path		
Name of user profile (output)		
Name of user profile source file (input)		

Data Types Used In Profiles

There are two types of data that may be written into a profile, but three ways to retrieve the data.

Data may be written to the profile as either a null-terminated string using the **PrfWriteProfileString** or as binary ASCII data using the **PrfWriteProfileData** call. For string writes, all characters up to and including the required null-terminator are written into the profile as the customization string. For binary data writes, the number of bytes specified in the call are written into the profile as the customization string. Null-termination characters are not used when writing binary data.

The difference between an ASCII string and ASCII binary data is more in how PM interprets the data than how it is stored in the profile. Beyond the null-termination character, there is no difference in the actual bytes inserted into the profile. For example, here are two entries from the sample program. The first is the Main window title written as a string. The second is a window color button ID written as ASCII binary data. The examples are shown both in decimal and hexadecimal notation for clarity.

(Dec) C O B O L / P M S a m p l e P r o g r a m
 (Hex) 434F424F4C2F504D2053616D706C652050726F6772616D00

(Dec) 0 7 0 5
 (Hex) 30373035

Data may be retrieved from a profile in one of three ways. The **PrfQueryProfileString** call returns the traditional null-terminated customization string, the **PrfQueryProfileData**

returns ASCII binary data and the PrfQueryProfileInt call returns the integer value of the string or binary data. The PrfQueryProfileString call may be used to retrieve string data only, while the PrfQueryProfileData and PrfQueryProfileInt calls may be used to retrieve data written using either the ASCII string or ASCII binary data format. The PrfQueryProfileString and PrfQueryProfileData calls will return whatever characters are found in the customization string, but the PrfQueryProfileInt call will return only the numeric value of a customization string that contains both alphabetic and numeric values.

Processing User Profiles

The system profile (OS2SYS.INI) and the default user profile (OS2.INI) appear as parameters in the PROTSHELL statement of the CONFIG.SYS file causing PM to open both files during system startup. Both files become a PM resource and remain open until system shutdown. As a result, applications can access them only via the profile processing PM API calls. As open PM resources, these two profiles have predefined handles, HINI-SYSTEMPROFILE for the system profile OS2SYS.INI, HINI-USERPROFILE for the default user profile OS2.INI or HINI-PROFILE file to refer to both profiles with one call. Application profiles, on the other hand, are application resources and, like any DASD file, must be opened by the program before use and closed prior to the end of the program. But unlike DASD files, opening a profile turns it into a PM resource. Opening the profile generates a handle that is used in profile processing in place of one of the default system handles. With the exception of the open and close calls, user profiles are processed in an identical manner to the system profiles.

Coding The PrfOpenProfile And PrfCloseProfile Calls

Before an application profile can be accessed, it must be opened using the PrfOpenProfile call. This call requires the variable holding the Anchor-block handle as parameter one. Parameter two is a pointer to the fully qualified profile name. Parameter three is the long integer that will receive the profile's handle when the call completes. For consistency, this variable should have, but does not require, a name starting with Hini-. For the sample program, this variable has the name *Hini-MyProfile*. Here is the declaration of the fully qualified profile name and the PrfOpenProfile call used in the sample program:

```

006410      Call OS2API 'PrfOpenProfile' using
006420                      by value      hab
006430                      by reference ProfileName
006440                      returning     Hini-MyProfile

```

Before the PM relationship is terminated, via the WinTerminate call, the open application profile must be closed via the PrfCloseProfile call. There is no requirement that a profile be closed as soon as its updating is complete, only that it be closed before the PM relationship is ended. When using a profile, I place the call to close it within the program's PM termination routine, where it will always be executed. Closing an unopened profile will not cause a problem, only return an *invalid handle* return code from PM.

The PrfCloseProfile call requires the variable holding the application profile's handle as parameter one and the standard return code variable as parameter two. Here is the PrfCloseProfile call used in the sample program:

```

008120      Call OS2API 'PrfCloseProfile' using
008130                      by value     Hini-MyProfile
008140                      returning   ReturnData

```

The Sample Program's Startup Processing

With this understanding of system profiles, it is now possible to check for the availability of the Courier font before the Main window is displayed, a task I skipped in Chapter 12. If the Courier image font is installed as a public font, there will be an entry in the default user profile, OS2.INI, under the application name of PM_Fonts and the key of COURIER. Querying the default user profile using this application and key name combination will return the fully qualified path and file name of the Courier image font if it is installed. If the font is not installed, the application has the option of continuing and using a substitute font, or displaying an error message asking that the Courier image font be installed before the program is run.

Coding The WinUpper Call

I talked earlier about the case sensitivity of the application name and key strings. For this reason, the WinUpper call is used in the sample program to ensure that the key has the required upper case form. The WinUpper call converts an entire string of lower or

mixed case characters into upper case characters. This call is required as the Courier string is stored in mixed case for use with the font metrics, but must be in upper case for the profile key name.

To code the WinUpper call, code the variable holding the Anchor-block handle as parameter one. Parameters two and three specify unique code page and country codes to use in the conversion. Coding both of these parameters as short integer nulls indicates that the existing code page and country code should be used. Parameter four is the pointer to the null-terminated string to be raised to upper case. If the string must be kept in its original case, be sure to move it to a work variable before issuing the WinUpper call. The last parameter is the variable that will receive the return code from the call. Here is the WinUpper call user to ensure that Courier is all upper case:

```
005960      Move Fattr -Name to KeyName
005970      Call OS2API 'WinUpper' using
005980          by value     hab
005990          by value     ShortNull
006000          by value     ShortNull
006010          by reference KeyName
006020          returning    ReturnData
```

Coding The PrfQueryProfileString Call

Remember, all profile customization strings are indexed under an application name and key. So before a query can be made, null-terminated string constants defining the application and key names must be declared. As an alternative, these strings could be stored within the resource file, then loaded into the program variables before the Prf calls are made.

To code the PrfQueryProfileString call, code one of the standard profile file handles or the variable holding the handle of an application profile as parameter one. The sample program queries the default user profile, OS2.INI, so the standard handle HINI-USERPROFILE is coded as parameter one. Parameter two is a pointer to the application name null-terminated string. Parameter three is a pointer to the key name null-terminated string. Parameter four is optional. If used, this parameter contains a pointer to a null-terminated string that will be returned if the requested customization string is not found. Use of the default string ensures that a string will always be returned, eliminating the need to check the return code. If not used, this parameter is coded as a long integer with a null value. When an optional string is not specified, a return code, the length of the returned string, of 0 indicates a *string not found* condition. Parameter five is a pointer to the buffer that is to receive the returned customization

string. The buffer must be equal to the maximum number of characters to be returned plus 1, the null-termination character. Parameter six is the maximum length string that may be returned and must not be larger than the buffer length specified as parameter five. The last parameter is the variable that will receive the length of the returned string. Here is the PrfQueryProfileString used to check for the availability of the Courier font:

```
006190 Call OS2API 'PrfQueryProfileString' using
006200           by value      HINI-USERPROFILE
006210           by reference PMFont
006220           by reference KeyName
006230           by value      LongNull
006240           by reference ProfileString
006250           by value      51 size 4
006260           returning    ReturnData
```

Because the sample program is only interested in the existence of the Courier image font, the returned string is ignored. A return code greater than 0 is sufficient to indicate the availability of the Courier image font. If no Courier image font entry was found in the profile, a message box is displayed telling the user to install the Courier font before running the sample program. Upon return from the message box call, the *program-done* flag is set to true and the program terminated. For additional information on using the WinMessageBox call, see Chapter 6. See Figure 14-3 for a picture of the Courier not found message box.

Querying For The Sample Program's Customization Strings

The sample program updates the entries in the SAMPLE.INI profile if the user enters a title for the Main window, specifies a color for the Main window's Client window or a

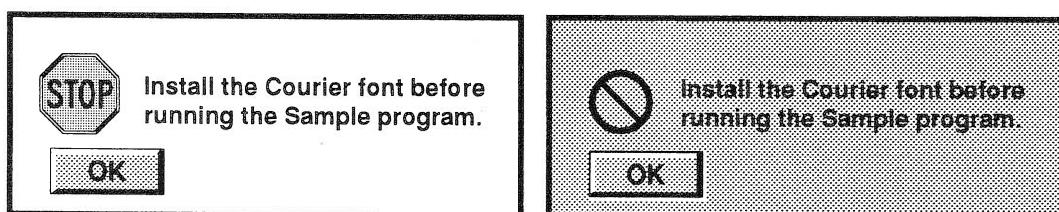


Figure 14-3 The missing Courier image font error message
(Version 1.3 - left, Version 2.0 - right)

color for the child windows. If either window color was selected, the last window color radio button selected is saved in the profile. These customization strings are saved under the application name *COBOL-PM*, with the keys of *ScreenTitle*, *MainColor*, *ChildColor* and *ID-Button*.

After determining that the Courier font is installed, the sample program's startup routine opens the user profile and receives the profile's handle, *Hini-MyProfile*. This handle is required by all of the queries and updates that will be performed during the program's execution.

With the user profile open and the profile's handle saved, the startup routine checks to see if a Main window title has been saved by querying the user profile with the application name set to *COBOL-PM* and the key set to *ScreenTitle*. This call is coded exactly the same as the prior PrfQueryProfileString call with the following parameters changed.

- Parameter 1 - The handle of the user profile
- Parameter 2 - The application name "PM-COBOL"
- Parameter 3 - The key name "ScreenTitle"
- Parameter 5 - A pointer to the screen title buffer
- Parameter 6 - A maximum return size of 31, 30 characters plus the null-terminator

Unlike the Courier image font query, if a title was saved, the startup routine needs to recover the returned string. A check for ReturnData greater than 0 indicates that a string was returned, and the following WinSetWindowText call sends the text to the Main window's Frame window. For additional information on coding the WinSetWindowText call, see Chapter 11. Here is the WinSetWindowText call used to send the program's title to the Main window's Frame window:

```
006570      If ReturnData > 0
006610          Call OS2API 'WinSetWindowText' using
006620              by value      hwndFrame
006630              by reference WEMainTitle
006640              returning    ReturnData
```

Following the setting of the Main window text, the startup routine queries for the Main window color.

Coding The PrfQueryProfileInt Call

Before the Main window can be displayed, the profile must be queried to determine if the user has specified a color for the Main window's Client window. The possible PM colors have values from -2 to 15, and are treated as signed short integers by the program. It is important to note that because the window color values are stored as binary data, negative color values can't be stored as true color values. Therefore, the startup routine must convert the positive representation of the color white back to its PM value of -2.

The PrfQueryProfileInt call has a structure that is very similar to the PrfQueryProfileString. Parameter one is the profile handle. For the sample program this is the handle of the user profile stored in the variable Hini-MyProfile. Parameters two and three are pointers to the application name and key index strings. Parameter four is the default integer value to be returned if the application and key combination cannot be found. Unlike the PrfQueryProfileString call, the value specified in parameter four will always be returned if the application and key name combination can not be found. To allow the program to correctly identify a *not found* condition, you must ensure that parameter four always contains a value that will not occur within the customization string. It is also important to note that PM will return a zero if the customization string is not numeric. The last parameter is the variable that will contain the returned integer upon completion of the call. Here is the PrfQueryProfileInt call to retrieve the Main window color:

```
006700      Call OS2API  'PrfQueryProfileInt' using
006710          by value    Hini-MyProfile
006720          by reference ProgramINI
006730          by reference MainWindColor
006740          by value    ShortNull
006750          returning   ReturnData
```

If the returned value is greater than zero, a check is made for a value greater than 15. A value of 16 is the color white, which must be converted back to its correct PM color value before being used by the WinFillRect call. The correct color value is then moved to *WEMWindowColor*, where it will be used by the Main window paint routine when the WM-PAINT message is received. If zero is returned, no action is taken and the default color of *WEMWindowColor* is preserved. Here is this routine for the Main window color:

```
006770      If ReturnData > 0
006810          If ReturnData > 15
006820              Compute ReturnData = ReturnData - 18
006830          End-if
006840          Move ReturnData to WEMWindowColor
006850      End-if
```

Following the retrieval of the Main window color, the profile is again queried for the color value of the child windows, using the PrfQueryProfileInt call. This call is identical to the Main window color query except for the key name *ChildColor*.

With the child window color retrieved, the color to be used for text drawing is calculated. Black is assumed. The variable *WECWindowColor* is tested for one of the six dark colors. If one of the darker colors is detected, *WETextColor* is changed to white.

As the last query, the button identity of the last button selected in the prior iteration of the dialog is extracted from the profile. As with the window colors, the profile is again queried using the PrfQueryProfileInt call for the value of the control button resource ID, *WE-BN-Color*. This query is identical to the prior PrfQueryProfileInt calls with only the key name changed to *ID-Button*. If a value is returned, it is used to set the default button upon entry into the next iteration of the Window Edit dialog. If a 0 is returned, indicating no value has been saved, then the default white button is retained.

With the retrieval of the button ID, all the stored user values have been accounted for and the Main window is displayed to the user.

Writing To A Profile

Adding or updating customization strings is very similar to querying customization strings. The program sets the application name string and key name string search arguments, sets a pointer to the customization string, and issues the appropriate profile write call. If the application and key name combination exists, PM will replace the related customization string with the new customization string. If the application name and key name combination do not exist, PM will create a new entry using the application name and key specified.

It is important to point out that there can be no invalid application name and key name combinations on a profile write, so there is no default value specified with the profile write calls. If the specified combination does not exist, a new application name and key

name entry are created. As a result, you must be absolutely sure of the application name string and key name string specified, including case sensitivity, when performing a profile write.

Updating the application profile in the sample program requires modifying the Window Edit routine by inserting profile write calls at appropriate locations within the existing data edit routine. As each user input is edited and implemented, it is written to the application profile under the *COBOL-PM* application name and related key name. Saving the new Main window title uses the PrfWriteProfileString call. Saving the integer values for the window colors and button ID uses the PrfWriteProfileData call.

Coding The PrfWriteProfileString Call

Upon returning from the Window Edit dialog routine, and after checking for the cancel button, validating that a new Main window title was entered and updating the Main window title, the PrfWriteProfileString call is issued to save the title in the application profile.

The PrfWriteProfileString call has the same first three parameters as the query calls, the handle of the profile and pointers to the null-terminated application name string and key name string. Parameter four is a pointer to the null-terminated customization string to be written to the profile. The last parameter is the variable that will receive the call's return code. Here is the PrfWriteProfileString call used in the sample program to write the Main window title to the profile:

```
012010      Call OS2API 'PrfWriteProfileString' using
012020              by value      Hini-MyProfile
012030              by reference ProgramINI
012040              by reference ScreenINI
012050              by reference WEMainTitle
012060          returning      ReturnData
```

Coding The PrfWriteProfileData Call

Following the writing of the Main window title, the window colors are verified and written to the application profile using the PrfWriteProfileData call. This call writes ASCII binary data to the profile, rather than a character string. This binary data write is required if the strings are to be queried with the PrfQueryProfileInt call and returned as integer values.

Before the color integers can be written to the profile, they must be converted to string values even though they will be written as ASCII binary data. This conversion is handled by COBOL when the integer value is moved to the string work area. In addition, if the color white was chosen as either window color, it must be converted to a positive number for storage in the profile as binary data.

To code the PrfWriteProfileData call, place the variable holding the handle of the profile in parameter one. As with other profile calls, Parameters two and three are pointers to the null-terminated application name and key name strings. Parameter four is a pointer to the converted binary data held in the string work area. This data must not have a null-termination character as it is not considered a string, in the C language sense of the word. Lacking a termination character, the length of the binary data to be written must be specified as an unsigned, long integer and it is coded as parameter five. The last parameter is the standard return code variable. Here is the PrfWriteProfileData call used to insert the Main window color into the user profile. The color button and child window color profile writes are identical, with only the key name and the customization string pointers changed.

```
012370      If WEMWindowColor = -2
012380          Move 16 to WorkString
012390      Else
012400          Move WEMWindowColor to WorkString
012410      End-if
012420          Call OS2API 'PrfWriteProfileData' using
012430              by value      Hini-MyProfile
012440              by reference ProgramINI
012450              by reference MainWindColor
012460              by reference WorkString
012470              by value      4 size 4
012480          returning    ReturnData
```


Chapter 15

Window Subclassing

When an existing window class procedure performs most of the tasks that your program requires or performs all of the tasks that your program requires, but not in the way your program requires, you can modify the functions performed by that window class procedure to meet your program's specific requirements through the process called window subclassing. Window subclassing is the creation of a new and slightly modified window class, a subclass, by intercepting the messages intended for the original window class procedure, processing those messages that require a change and returning all other messages to the original procedure.

For example, in this chapter I want to replace the frame resizing, double-headed arrow pointer normally displayed by the default frame window procedure with a custom frame-grabbing hand similar to the hand pointer currently displayed by the Client window. To accomplish this, I could write my own frame window procedure. But, as a replacement for the existing frame window procedure, my procedure would be required to duplicate all of the tasks currently performed by the default Frame window procedure (sizing, moving, maximizing and minimizing etc.) in addition to the single mouse pointer function that I would like to change. Since I only wish to change the mouse pointer function, a better choice is to subclass the default frame window procedure. Through subclassing my procedure intercepts all the messages intended for the frame window procedure, processes only the WM-MOUSEMOVE message and returns all other messages back to the default frame window procedure for normal processing. By utilizing window subclassing, the sample program gets the effect of a customized frame window procedure without having to duplicate all the functions of the default frame window.

The subclassing program does not alter the window procedure that it subclasses. Indeed, that is not possible. Instead, subclassing allows a program to intercept the messages destined for the original procedure and redirect them to a new procedure within the application. The new procedure then processes those messages that require changes and

passes back to the original procedure those messages that require no modifications. See Figure 15-1.

While the intercepted messages are within the subclassing procedure, they may be manipulated in any way desired. Messages that are processed by the subclassing procedure are, of course, not returned to the subclassed procedure. But for those being passed back to the original procedure, the subclassing procedure may pass on the message exactly as received, change the contents of the message or substitute another message for the message received. The choice of which action to take depends upon the extent of the original procedure's functions that are to be preserved.

Implementing Window Subclassing

Window subclassing is implemented by writing a window procedure that performs the functions of the original procedure that require modification or replacement, or functions that are to be logically added to the original procedure. The address of the original window procedure is then replaced with the address of the subclassing window procedure using the WinSubclassWindow call. This call returns the address of the original window procedure, which becomes the entry point called for all messages to be returned for default processing.

The messages to be passed back to the original procedure are not returned to PM using the WinDefWindowProc or sent using the WinSendMsg or WinPostMsg calls as you might expect, but are passed directly to the original procedure using a standard COBOL inter-module call with the stack arranged according to the Pascal calling convention. Here is an example of the call required to pass a subclassed message back to the original window procedure. In this example, the variable WindowProc is declared as a procedure pointer and contains the entry point address of the original window procedure. Notice two requirements for the call that returns subclassed messages. First, the OS2API phrase is required to generate the Pascal calling stack structure and second, *WindowProc*, as a variable holding an address, is not contained with quotation marks, as are normal PM calls.

```
Call OS2API WindowProc using
      by value  hwnd
      by value  Msg
      by value  MsgParm1
      by value  MsgParm2
      returning Mresult
```

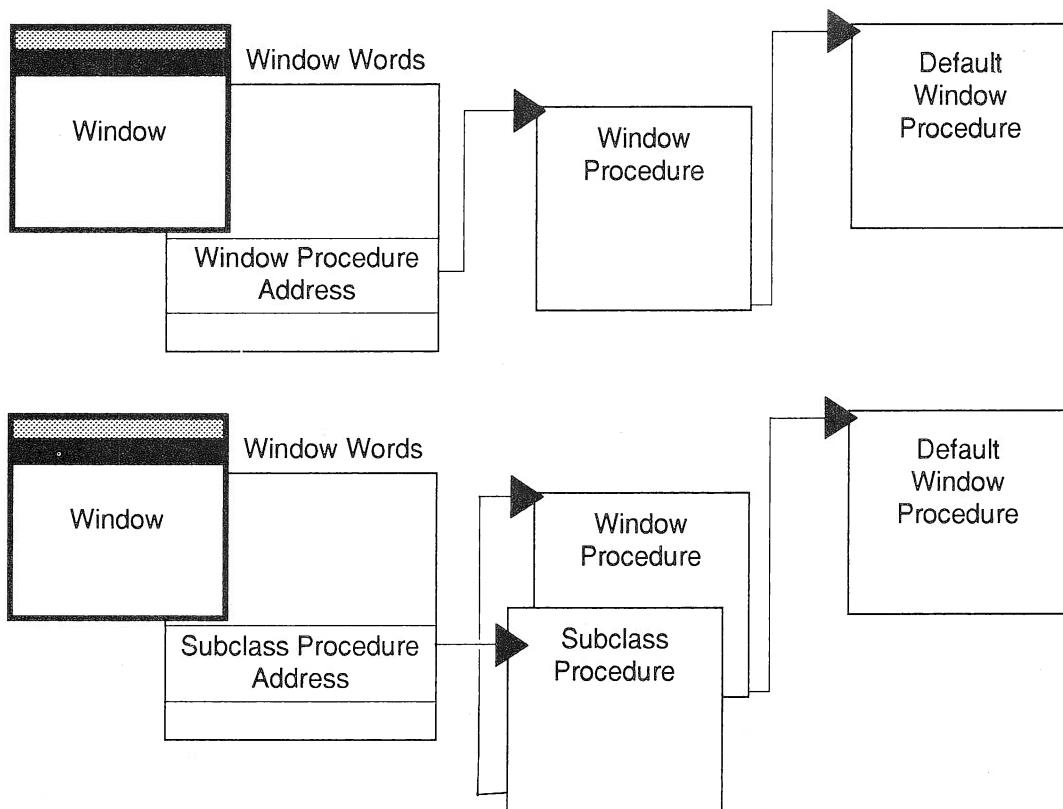


Figure 15-1 Linking of windows to window procedures, before and after subclassing

The subclassing of a window procedure can be reset at any time by simply reversing the addresses. Issue the WinSubclassWindow call a second time with the procedure address returned in the first call as the address in parameter one of the second call.

Subclassing A Single Window Or An Entire Class

A program has the choice of subclassing a single instance of a window or an entire class of windows. To subclass an individual instance of a window, the address of the window procedure to be subclassed is replaced with the address of the subclassing procedure after the window is created, using the WinSubclassWindow call. With only a single Main window, the sample program uses this individual window subclassing to manipulate the Main window's Frame window pointer.

When a window class supports several windows, or multiple iterations of a single window, it is usually more efficient to subclass the entire class of windows. To subclass an entire window class, register a new subclass procedure, using the same class name as the window class to be subclassed. The parameters passed with the new WinRegisterClass call will replace the existing parameters for the specified window class, including the new window procedure address.

A Word About Window Words

Every instance of a window has an accompanying area of memory where PM stores window-related information. This area of memory, allocated for each window and based upon the specifications of the window's class, is referred to as a window's window words. Every instance of every window has a number of window words reserved to store the following type of window-related data:

- A pointer to the window procedure
- The parent-window handle
- The owner-window handle
- The handle of the first child window
- The handle of the next sibling window
- The window's size and position
- The window's style
- The window's identifier
- The message queue handle

Programs can access this data through various Presentation Manager calls. For example, the window procedure address is available through the WinSubclassWindow call, while the size and position of a window is available through the WinQueryWindowPos call. A Program can specify an additional amount of storage be set aside for its own use. The amount of additional window words storage to be allocated is specified for a class as part of the WinRegisterClass call and applies to every window created with that class.

Window words are an extremely powerful feature of PM. The use of window words allows a program to store data unique to each instance of a window along with that window, then retrieve and make use of the data without having to manage the relationship between the data and its related window. In addition, because window words are a PM resource with data storage and retrieval is done through PM, addressability to window words does not need to be established, as it must when passing data between procedures.

As with resource data, the application simply passes a pointer to an integer variable declared within its address space, and PM returns the data in this variable.

In Chapter 11, I introduced dialog data passing and the problem saving the structure pointer during WM_INITDLG message processing. You can see now that the single window word, QWL-USER, available with every dialog frame, is the correct place to store the pointer, not back in Working-Storage as I did then to avoid the topic of window words. Any window-related data really belongs in a window word.

But, as with so much of the PM resources that are made available to your programs, you must use window words very sparingly. Remember, defined window words will be replicated for each window created with that class, and PM resources are not inexhaustible. Normally, window words are defined for no more than one or two values. Rather than storing all window related variable data within window words, you should dynamically create a data structure for instance of a window within allocated memory and store a pointer to the related data structure within the window words.

Application-defined window words can be created for any window class using the WinRegisterClass call. Parameter five of WinRegisterClass defines the number of storage bytes to be allocated for each window instance created with the associated class. The data structure of these words, whether long integer or short integer, and their contents, is entirely up to the application.

Window words are a block of contiguous storage allocated for each instance of a window within a class. The definition of the allocated storage as long or short integers is made when the data is actually loaded into the block of storage. You may stuff as many long or short integers into the allocated space as possible, but you may not exceed the number of bytes allocated in the WinRegisterClass call.

Window words are addressed using a zero-based index representing the location of each word within the block of storage. The index for application-defined window words starts at 0 and runs through the number of extra bytes allocated minus a constant. For long integer words, the indices run up to the number of bytes allocated minus four. For short integer words, the indices run up to the number of bytes allocated minus two. For

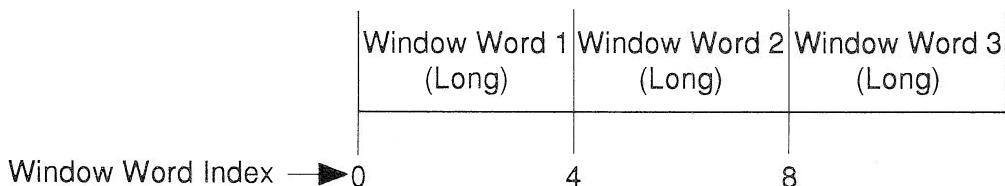


Figure 15-2 Window word indexing

example, if your application allocated 12 bytes of window words, these words would have indices of 0, 4 and 8 (12 - 4) for long integer words or 0, 2, 4, 8, 10 (12 - 2) for short integer words.

Window words are queried using either WinQueryWindowULong for long integers or WinQueryWindowUShort for short integers. The actual value returned is determined by the zero-based index value passed as part of the call. Applications generate the index values that control access to the data within their window words. For system window words, predefined index values have been established to allow applications to query this data. The following list shows the predefined index values for the most common system window words. Those labels that start QWS- are used only for short integers and only with the WindowUShort calls. Those labels that start QWL- are used only for long integers and only with the WindowULong calls.

Window Word Index Values

QWS-ID	The window identity from the WinCreateWindow call.
QWS-USER	A short integer reserved for application use.
QWS-FLAGS	Flags showing the current status of frame and dialog windows.
QWS-RESULT	The dialog results as established by the WinDismissDlg call.
QWS-XRESTORE	The x coordinate for the position to which the window will be restored.
QWS-YRESTORE	The y coordinate for the position to which the window will be restored.
QWS-CXRESTORE	The width to which the window will be restored.
QWS-CYRESTORE	The height to which the window will be restored.
QWS-XMINIMIZE	The x coordinate for the position to which the window is minimized.
QWS-YMINIMIZE	The y coordinate for the position to which the window is minimized.
QWL-HMQ	The handle of the window's message queue.
QWL-STYLE	The window style.
QWL-USER *	A long integer reserved for application use.
QWL-HHEAP	The heap handle used by child windows of this window.
QWL-HWNDFOCUSSAVE	The window handle of the child window that last held focus when this frame was deactivated.
QWL-DEFBUTTON	The default push button for a dialog.

* Only the predefined classes of WC_CONTAINER, WC_FRAME (including dialog windows), WC_LISTBOX, WC_BUTTON, WC_STATIC, WC_ENTRYFIELD, WC_SCROLLBAR and WC_MENU have the additional long integer word, QWL_USER available for use by the application.

Creating The Frame-Grabbing Pointers

Two new pointers are required to support the modifications made to the sample program in this chapter. One pointer, *ID-LHandptr*, represents the hand that will grab and move the vertical frame borders left or right. The second pointer, *ID-DHandptr*, represents the hand that will grab and move the horizontal frame borders up or down. Both of these pointers are derivatives of the original hand pointer developed in Chapter 8. If you are unfamiliar with creating pointers using the Icon Editor, please review Chapter 8. The detailed design for both of these new pointers is shown in Figures 15-3 and 15-4.

Making Changes To The Sample Program

The addition of two new pointers will require new pointer entries in the Resource Script file with corresponding entries in the Resource Header file. These entries are identical to the pointer entries made in Chapter 8, with only the pointer names changed. For more information on adding pointer entries to the Resource Script file, please review Chapter 8.

In addition to updating the Resource Script file, a change is required to the WinRegisterClass call used to register the window class of *MainWinClass*. The implementation of window subclassing requires the existence of application-defined window words for the Main window's Frame window. To reserve this space, parameter five of the WinRegisterClass call is replaced with the unsigned short integer variable *WindowWords* holding the number of bytes to be reserved. Here is the revised WinRegisterClass call with the changed parameter five:

```

002600 77 WindowWords          pic 9(4) comp-5 value 8.

005200      Call OS2API 'WinRegisterClass' using
005210          by value     hab
005220          by reference MainWndClass
005230          by value     WindowProc
005240          by value     CSClass
005250          by value     WindowWords
005260          returning   ReturnData

```

Also required are the addition of two WinLoadPointer calls to load the two frame-grabbing hand pointers from the resource file. These calls are identical to the existing WinLoadPointer call used to load the hand pointer with the exception of parameter

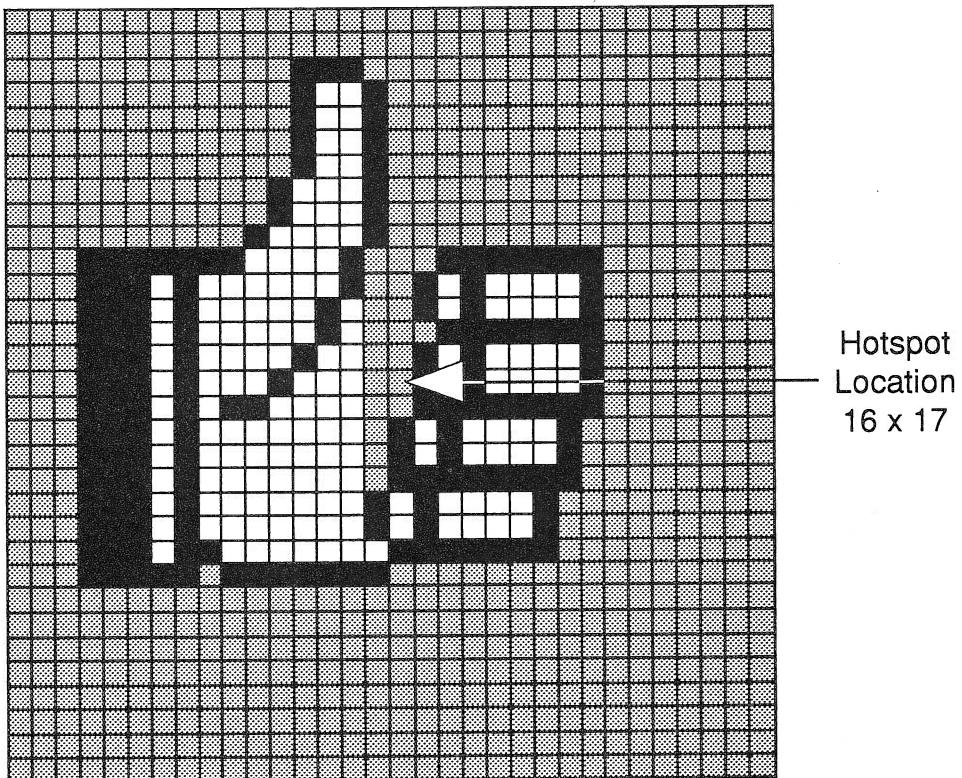


Figure 15-3 The vertical frame grabbing hand pointer

three, which reflects each pointer's unique name. These additional calls were inserted immediately after the existing WinLoadPointer call.

Coding The WinSubclassWindow Call

The implementation of individual window subclassing requires only the loading of the entry point address of the subclassing procedure followed by the WinSubclassWindow call. For COBOL programs, the address of the subclassing procedure must be loaded into a variable defined as a procedure pointer using the COBOL Set statement.

To code the WinSubclassWindow call, place the variable holding the handle of the window whose procedure is to be subclassed in parameter one. For the sample program I want to subclass the Main window's Frame window, so parameter one is coded as *hwndFrame*. Parameter two is the variable holding the address of the subclassing

window procedure, while parameter three is the variable that will receive the address of the procedure being subclassed. For the sample program, the returned address will be the address of the Main window's Frame window default procedures. The use of the same variable for passing and receiving different pointers is allowed in this call, since once the new subclass procedure address is passed to PM the contents of the variable are no longer required, and the variable can receive the returned procedure address. Here is the WinSubclassWindow call used in the sample program to subclass the Main Window's Frame window:

```
005730      Set Windowproc to ENTRY 'FrameProc'
005740      Call OS2API 'WinSubclassWindow' using
005750          by value hwndFrame
005760          by value WindowProc
005770          returning WindowProc
```

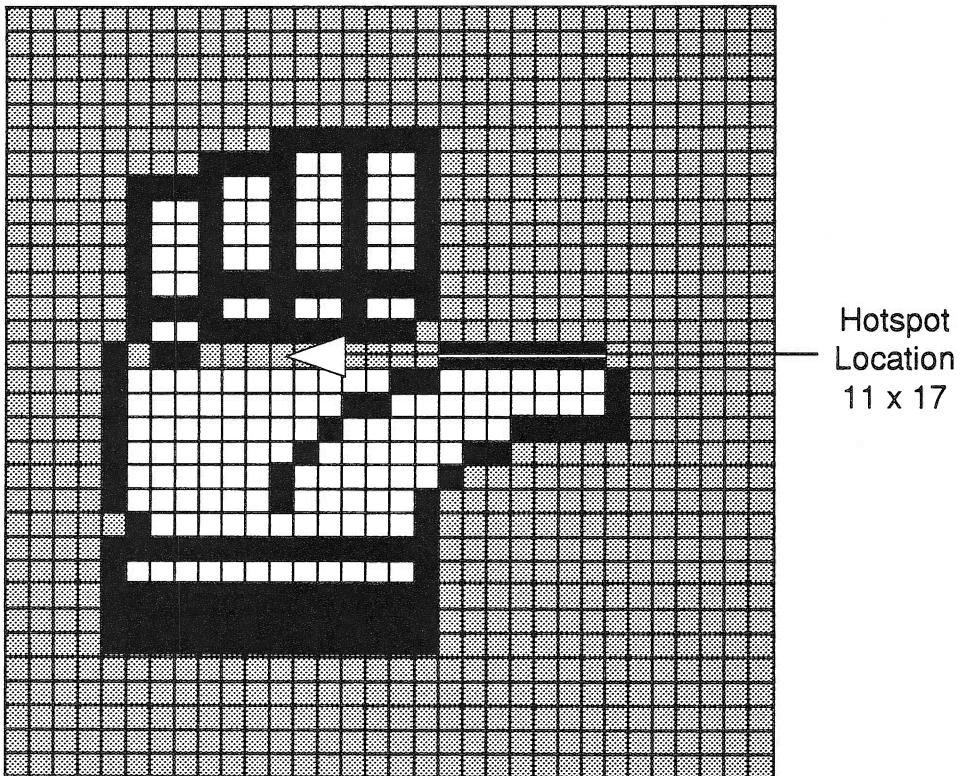


Figure 15-4 The horizontal frame grabbing hand pointer

Loading The Window Words

After making the WinSubclassWindow call, the sample program stores two pieces of information in the Frame window's window words that are required by the subclassing procedure. Using the WinSetWindowULong call, the sample program saves the subclassed frame window procedure address, returned by the WinSubclassWindow call, in the first application window word. This address is required by the subclassing procedure to pass unwanted messages back to the default frame procedure. The second WinSetWindowULong call stores the handle of the window whose frame is being subclassed in the second window word. The use of these two variables will be discussed in the following section.

To code the WinSetWindowULong call, place the variable holding the handle of the window whose window words are to be loaded in parameter one. Parameter two is the variable holding the zero-based index into the window words where the value will be stored: 0 for the default frame window procedure address and 4 for the handle of the Main window's Client window. Here is the WinSetWindowULong call used in the sample program to store the default frame window procedure's address into the window words at index 0:

```

002090 77 QWL-OldProc      pic s9(4) comp-5 value 0.

005810      Call OS2API 'WinSetWindowULong' using
005820          by value hwndFrame
005830          by value QWL-OldProc
005840          by value WindowProc
005850          returning ReturnData

```

The Subclassing Window Procedure

The structure of a window procedure that performs the subclassing function is identical to all other window procedures, with the exception of the way messages are passed back to the subclassed procedure. Any message that is not processed by the subclassing procedure must be sent directly to the procedure that is being subclassed, not to the PM default processing procedure, as is normally the case. In other window procedures, unprocessed messages are returned to PM via the WinDefWindowProc call. In a subclassing procedure, unprocessed messages are returned to the subclassed procedure by calling the entry point address of the procedure, with the individual message parameters as the passed data.

Coding The WinQueryFocus Call

The function of the sample program's subclassing procedure is to display a custom frame-grabbing pointer when the mouse pointer is moved over the Main window's Frame window. All other functions of the default frame window procedure are compatible with the sample program and do not require processing. As a result, the message processing routine consists solely of intercepting and processing the WM-MOUSEMOVE message. The COBOL Evaluate statement contains only the *when WM-MOUSEMOVE* and *when other* phrases.

The primary functions of the WM-MOUSEMOVE message processing involves determining if the sample program's Main Window has the focus and, if so, where on the frame the mouse pointer is located. The pointer's frame location determines which custom pointer is used. It is important to note that WM-MOUSEMOVE messages are sent to the sample program's window procedures whenever the mouse moves across any part of a sample program window, regardless of where the focus lies. But, when a window does not have the focus, it is incorrect to change the pointer, as this implies that the window has the focus when, in fact, it does not.

So, upon receipt of a WM-MOUSEMOVE message, the subclassing procedure ascertains which window has the focus. The WinQueryFocus call is used to obtain the handle of the window that currently holds the focus. This handle is then compared to the handle in the second window word to determine if the Main Window currently has the focus.

To code the WinQueryFocus call, use the standard definition of the desktop as parameter one. Parameter two, the window lock state, is no longer used by OS/2 and must be coded as a null. Parameter three is the variable that will receive the handle of the window currently holding the focus. Here is the WinQueryFocus call used in the subclass procedure:

```

008540      Call OS2API 'WinQueryFocus' using
008550          by value HWND-DESKTOP
008560          by value ShortNull
008570          returning hwndFocusWindow

```

Coding The WinQueryWindowULong Call

The handle of the window whose frame is being subclassed is stored in the second word of the Main window's Frame window's window words. To obtain this handle, stored as

a long integer using the WinSetWindowUlong call, the WinQueryWindowULong call is used. To code this call, place the variable holding the handle of the window whose window words are to be queried in parameter one. Window words are a property of frame windows, so the handle used here is the handle of the Main window's Frame window. Since the subclassing procedure is processing messages intended for that window, the current message handle can be used as parameter one. Parameter two is the zero-base index of the window word to be queried. The window handle is the second of two long integers stored in the window words, so an index value of 4 is used. The last parameter is the variable that will receive the returned handle. Here is the WinQueryWindowULong call used to extract the Client window's handle from the window words:

```

002100 77 QWL-hwndWindow      pic s9(4) comp-5 value 4.

008590          Call OS2API 'WinQueryWindowULong' using
008600              by value    hwnd
008610              by value    QWL-hwndWindow
008620              returning   hwndMyWindow

```

If *hwndFocusWindow* = *hwndMyWindow*, then the Main window has the focus and one of the frame-grabbing pointers is loaded. To determine which custom pointer to use, the procedure must determine the current location of the mouse pointer. The logic used determines if the mouse is over the top or bottom horizontal frame and, if not, a vertical frame location is assumed. Since this message is sent to the frame window procedure only when the mouse is over the frame window, this assumption can safely be made. The WinQueryWindowRect call returns the size of the frame window. The WinQuerySysValue call, using the SV-CXSIZEBORDER system value ID, returns the width of the border. Subtracting the border width from the top of the window gives the range of pixels displaying the top horizontal frame, and adding the border width to the bottom of the window gives the range of pixels displaying the bottom horizontal frame. If the Y coordinate of the mouse, the second short integer of the WM-MOUSEMOVE message's parameter one, is within either of these two bands, the top/bottom frame-grabber pointer is loaded. If it is not within either of these two bands, the side frame-grabber pointer is loaded. Once the correct pointer is determined, the WinSetPointer call is made to PM.

Sending Messages On To The Original Procedure

All other messages intercepted by the subclassing procedure are returned to the Main window's default frame window procedure, the original destination of the messages. The messages are not sent to PM using the normal WinDefWindowProc call; instead, the entry point of the subclassed procedure is called directly, using a standard COBOL inter-

module call with the message parameters passed on the stack as a series of values. Passing messages in this manner causes the subclass procedure to be suspended until the subclassed procedure returns control at the end of its message-processing routine. This direct call to the subclassed procedure points out a major consideration whenever subclassing is to be used. You cannot subclass a procedure that does not correctly return control to your calling procedure. If control is not returned to your subclassing procedure, it will remain suspended, unable to process incoming messages and eventually lock the entire application.

To make this call, the address of the frame procedure's entry point must be obtained. This is stored in the first of the two window words. As with the other window word, it is stored as a long integer, so it is extracted using the WinQueryWindowULong call. This call is coded exactly the same as the prior call, with only the window word index changed.

With the entry address in hand, a call is made directly to the subclassed procedure's entry point. To code this call, place the variable holding the procedure's address as the target of the call statement. Parameters one through four are the message parameters of the message that is to be passed back to the subclassed procedure. The last parameter is the variable that will receive the return code from the subclassed procedure. Here is the call to retrieve the entry point address of the subclassed procedure from the first application window word, followed by the call to pass an unwanted message back to the subclassed procedure.

```
008950      Call OS2API 'WinQueryWindowULong' using
008960          by value hwndFrame
008970          by value QWL-OldProc
008980          returning WindowProc

009000      Call OS2API WindowProc using
009010          by value hwnd
009020          by value Msg
009030          by value MsgParm1
009040          by value MsgParm2
009050          returning Mresult
```


Chapter 16

Dynamic Linking

Within the OS/2 and PM environment there are a range of programming functions from the very complex to the relatively simple. Somewhere between these two extremes there lies a set of functions that are perceived as complex but are, in fact, relatively easy to implement. Dynamic linking is certainly part of this middle group.

Dynamic linking offer powerful capabilities to programmers developing OS/2 and PM applications. The ability to defer the loading of a dynamically linked routine until it is needed, coupled with the ability to share a dynamically linked module's code and resources across the system, has a profound effect upon the way OS/2 and PM applications are developed. The use of dynamic linking can affect everything, from a program's memory requirements to its execution speed to how it is written and maintained.

If you have a hard time envisioning how dynamic link libraries might be used, or what impact they might have on application development and maintenance, consider the fact that every one of the almost 1,000 OS/2 Kernel and PM calls within the OS/2 system is a dynamically linked function. The ease with which these calls are invoked belies the complexity that they contain and the capabilities they offer to programs that use them. Yet, the same ease of use that dynamic linking brings to the implementation and maintenance of complex OS/2 and PM calls can be applied to all aspects of your application development. OS/2 and PM calls could just as easily be calls to common subroutines or functions developed by you for your own applications.

The impact would be immense if program development and maintenance had to manage OS/2 and PM calls in the traditional static linking environment. As an example, the sample program issues 55 different PM calls. The impact of having to statically link even these few PM calls into the sample program would be enormous. The Linker would require access to the object code for each of the 55 functions, making the link step complex and time-consuming. The size of the sample program, along with every other PM

program on your system, would surely double or triple, and any time one of these calls was changed, every copy of every PM program using the changed call would require relinking.

Dynamic linking is nothing more than an alternate way of packaging program code or resources. Like the eternal "paper or plastic" decision faced by all grocery shoppers, dynamic linking is the alternative to the standard program packaging. As a packaging decision, dynamic linking, in and of itself, has no effect on the way a program is written. It is true that very often dynamically linked modules are more complex than their statically linked cousins, but that has more to do with the tasks performed by dynamically linked programs than requirements imposed by dynamic linking. Any reentrant COBOL subroutine can be converted into a dynamically linked module with few or no modifications.

Dynamic linking is the process of delaying the resolution of a program's external references from the traditional link edit time to the time when the module is loaded. External references that are resolved as the module is loaded are said to be dynamically resolved, leading to the term dynamically linked modules. Delaying the resolution of a program's external references, from link edit time to program load time, allows functions that are not part of the module built by the Linker to be called when the program is executed. This means functions can be shared among all programs and the loading of individual modules, subroutines or resources can be deferred until their functions are required.

Dynamic link library is the name given to an executable module that contains one or more dynamically linked functions or subroutines.

Public And Private Dynamic Link Libraries

There are two classes of dynamically linked libraries, public and private. Public, or global, dynamically linked libraries are automatically loaded by OS/2 during system initialization, and are available to any program running in the system. PM and OS/2 Kernel calls are the prime example of public DLLs. Private dynamically linked libraries are loaded by individual programs and kept as a private subroutine of the loading program. Private DLLs are available only to the program that loads them. This, of course, does not prevent private DLLs from being shared by all programs that know of their existence. The distinction between public and private DLLs relates more to how they are installed and loaded, rather than who can use them.

To be classed as a public dynamic link library, the name of the DLL must be stored in the OS2.INI file under the application name of SYS_DLLS and the key name of LoadOneTime. An application installing a public DLL must include a

PrfWriteProfileString call within the install procedure to insert the name of the DLL into the SYS_DLL list within the OS2.INI file.

As each public DLL is loaded by OS/2 during system startup, it is initialized by PM calling the DLL's first entry point, defined as ordinal number 1. See the section later in this chapter for more information about the use of ordinal numbers.

Most private DLLs are not loaded until they are first called by a program. In these cases, no reference to the dynamic link library is made; calling the specific entry point is sufficient to cause the loading of the DLL. Where multiple code segments reside within the same DLL, calling an entry point of one code segment will cause only that code segment to be loaded. The remainder of the DLL's code segments will remain unloaded. Within the sample program, the Printer Setup Dialog module (PRTSAMPL) is automatically loaded when the call is made to the PRTSAMPL entry point.

But, for those modules that contain only resources and are not called directly, or for those modules that require preloading and initialization by the program before they can be invoked, the DosLoadModule, DosGetProcAddr and DosFreeModule calls must be used. When the DosLoadModule call is made, the entire DLL is loaded into memory, the calling program is linked to the loaded module and all external addresses are resolved. Within the sample program, the Printer Setup dialog template (PRTDLGBX) is never called directly by the Printer Setup Dialog so it must be loaded prior to the Printer Setup dialog using the DosLoadModule call.

The following OS/2 calls are used whenever a dynamically linked module must be loaded by a program, a module's entry points must be retrieved or a previously loaded module is to be freed after it is used.

DosLoadModule: This call loads a dynamically linked library into memory and returns the module's handle. If the module is already loaded into memory, then loading is suppressed and the module's handle returned. Alternately, the DosGetModHandle call could be used to determine if the module is already loaded, but I do not recommend the use of the DosGetModHandle due to the restrictions on the use of the returned handle with the DosGetProcAddr call.

DosGetProcAddr: This call returns the far address of an entry point within a dynamically linked module. Where the external name of a subroutine is not known, this call may be used together with the subroutine's ordinal number to obtain the entry point address. The DosLoadModule must be issued before this call to ensure that the module is correctly attached to the program. The handle returned by the DosGetModHandle may not be used with this call for reference to a module that is already loaded.

DosFreeModule: This call ends a program's use of a DLL. In fact, all this call does is reduce the DLL's reference count by one. When the reference count reaches 0 - all references to the module have been freed - the module is removed from memory.

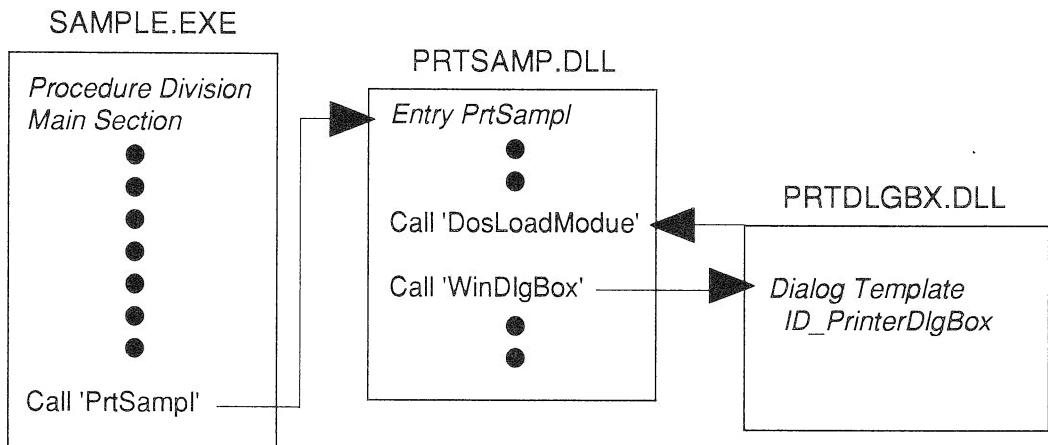


Figure 16-1 The sample program's relationship with the printer DLLs

Preload And Loadoncall Dynamic Link Libraries

The loading of a DLL can be deferred until the code, data or resource is required. But, this is not a requirement and DLLs containing code, data or resources that are used early in the program, or often throughout the program, may be loaded when the program is loaded. The decision of when to load a module should be made by the program's designers. It is a trade-off of the extra memory required to hold the module versus the time it takes to load the module and resolve its external references.

At the time a dynamic link library module is linked, the type of code and data segment loading must be defined. Dynamically linked routines may be defined as load when the program begins (PRELOAD), or load only when called by a program (LOADONCALL). Unless a segment is specifically marked in the Module Definition file as PRELOAD, it will automatically be defined as LOADONCALL.

Creating The Linker Response File

Modifications to the Linker Response file are required when linking a dynamic link library module. There are no major changes here; the modified Linker Response file should look familiar to the one you have been using to statically link the sample program. The Linker Response file specifies the name of the input object module, the name of the generated executable file, the name of the Linker map file, any Linker run options, a list of the libraries to be searched and the name of the Module Definition file to be used.

With allowances for the different module names, the Linker Response file used to build the Printer Setup dialog procedure DLL is identical to the sample program's Linker Response file, with one exception. The executable file name has the file extension of .dll, indicating a dynamic link library, rather than the usual executable file extension of .exe. Here is the Linker Response file to be used when linking the Printer Setup Dialog procedure DLL:

Version 1.3

```
prtsamp.obj  
prtsamp.dll  
nul.map  
lcobol.lib+  
os2.lib+  
pmcob.lib+  
pmbind.lib  
prtsamp.def  
/NOD /NOP  
;
```

Version 2.0 (16-bit)

```
prtsamp.obj  
prtsamp.dll  
nul.map  
lcobol.lib+  
os2286.lib  
prtsamp.def  
/NOD /NOP  
;
```

Creating The Module Definition File

As you can see, other than changing the file extension of the output file to .dll, there is nothing in the Linker Response file to cause the Linker to create a dynamic link library in place of a normal program. Actual control over what the Linker produces rests with the Module Definition file. The Linker Response file tells the Linker where to find the information that it needs, but it is the Module Definition file that tells the Linker the specific type of module to produce. For additional information about the Module Definition file, see Chapter 3 and the *Presentation Manager Programming References*, part of the OS/2 Developer's Toolkit.

The Module Definition file used to construct a dynamically linked module, or used to construct a module that will access a dynamically linked module, contains five entries that warrant a closer look. These entries are the most important when creating any dynamic link library.

The following Module Definition file used to create the Printer Setup dialog procedure DLL contains the line numbers referenced in the following descriptions.

Line 1	LIBRARY PRTSAMP DESCRIPTION 'OS/2 Print Dialog COBOL/2 DLL' PROTMODE
Line 4	CODE LOADONCALL
Line 6	DATA NONSHARED EXPORTS PRTSAMPL @1

Line 1**LIBRARY:**

The key word LIBRARY instructs the Linker to build a dynamic link library module rather than the standard executable file. The name specified as parameter one is used as the name of the generated DLL. The key word NAME, used in the sample program's module definition file, instructs the Linker to build a standard executable file. Library and Name are mutually exclusive entries.

Line 4**CODE:**

This line defines the code segment default attributes. LOADONCALL specifies that the code segment for the Printer Setup Dialog procedure DLL is not to be loaded into memory until the procedure is called. This is the first time in the sample program's development that the loading of code or data has been deferred until needed. Note that while the DATA line does not specify a load option, when LOADONCALL is specified for the code segments, LOADONCALL is assumed for the data segments.

It is important to note that COBOL subroutines using the compiler's statically linked run-time support will have the DLL's run-time support loaded when the main program is started, even though the loading of the user-written subroutine is deferred until a call is made. The loading of the compiler's run-time support cannot be controlled by the user-written DLL or its Module Definition file.

Line 6**EXPORTS:**

The EXPORTS entry defines the name and characteristics of each subroutine or function within this dynamic link library that is available for use by other applications. You should think of EXPORTS as a list of available entry points within the DLL. Every EXPORTS statement must have a corresponding COBOL entry point within the module. The EXPORTS keyword marks the beginning of the list of shared functions. Each function requires a separate line and the number of functions is limited to 3072. For the Printer Setup Dialog procedure DLL, there is only one COBOL entry point, PRTSAMPL, so only a single EXPORTS line is needed.

The name specified in the EXPORTS statement becomes the name by which this subroutine or function is known to all programs that call it.

This name may be either the actual name used in the COBOL entry statement, as with PRTSAMPL, or a completely different external name. To utilize the entry point name, simply code the entry point name as the first parameter of the statement. To utilize an external name, the first parameter in the EXPORTS statement must be coded as follows.

```
EXPORTS      entryname=internalname
```

Line 6

ORDINALS: There are two ways in which a subroutine or function may be called. The most common way is by calling the external name directly. However, it may also be called using an ordinal number. The ordinal number is the index number of the subroutine or function within the entry table of the dynamic link library module. Every module has an index number, whether it is assigned or not. There will always be, for example, a first entry in the table that is the entry with the ordinal of 1. The use of an ordinal number allows you to select the ordinal number rather than the Linker.

If the ordinal number is to be used to access the subroutine or function, the DosGetProcAddr call must be issued to obtain the entry point address of the subroutine or function to be called. For the Printer Setup Dialog procedure, I have specified an ordinal number of 1. Thus, the Printer Setup Dialog procedure can be called by its external name, PRTSAMPL, or the ordinal number of 1 can be used to obtain the entry point address via the DosGetProcAddr call.

Line 6

RESIDENTNAME: The use of the keyword RESIDENTNAME on an EXPORTS line causes the function's name to be kept in memory at all times. This keyword is valid only when an ordinal number is specified.

The following revised Module Definition file for the Sample program shows the line number referenced in the following descriptions.

```
NAME SAMPLE WINDOWAPI
DESCRIPTION 'OS/2 Presentation Manager COBOL/2 Sample
Program'
DATA MULTIPLE
PROTMODE
STACKSIZE 40960
HEAPSIZE 16384
IMPORTS PRTSAMP.PRTSAMPL
```

Line 7

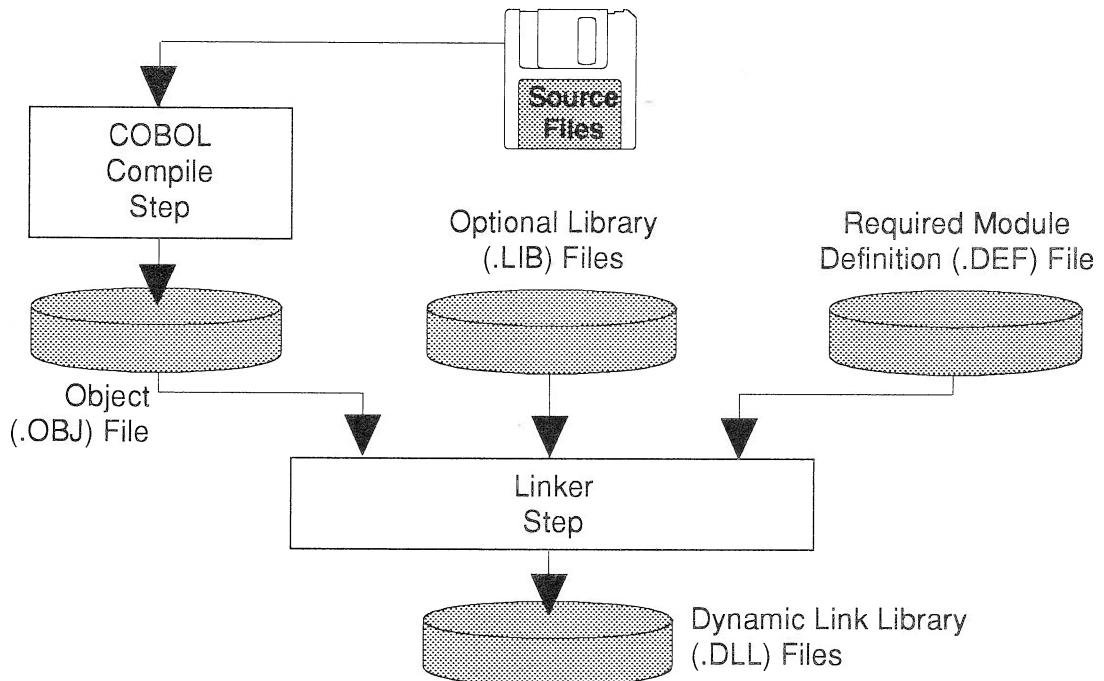


Figure 16-2 The steps to build a dynamic link library

Line 7

IMPORTS: Just as a dynamic link library must indicate which of its functions are available to other applications, programs that use these functions must indicate which of their functions are external to their code segments, and thus require that the external address resolution be deferred beyond link edit. This is done by either adding an IMPORTS statement to the Module Definition file, as is done with the sample program, or by including an Import Library in the Linker's list of search libraries, as is done for OS/2 and PM calls.

The IMPORTS statement identifies the external name of the subroutine or function whose addresses must be resolved by the Program Loader at load time. The IMPORTS keyword marks the beginning of the list followed by any number of IMPORTS statements, each requiring a separate line. Each IMPORTS statement must specify the dynamic link library module name first, followed by a period, followed by the subroutine or function name. Here is the format of the IMPORTS statement:

```
[internal-name=] libraryname.entryname
```

Building An Import Library

As an alternative to coding IMPORTS statements in the Module Definition file, you may use an import library. The import library resolves the external references by supplying pointers to the various functions contained within one or more dynamic link libraries. Import libraries are especially handy when a large number of IMPORTS statements must be coded, or the DLL is used by a large number of programs. Prime examples of the use of import libraries are the OS2.LIB, OS2286.LIB and OS2386.LIB libraries used to resolve the external OS/2 and PM calls during link edit. To highlight the advantages of import libraries, the sample program would need to code each of its 55 OS/2 and PM calls as separate IMPORTS statements, if the OS2.LIB, OS2286.LIB or OS2386.LIB import libraries was not available.

Once created, the import library must be placed in a directory within the library search path, and the name of the import library added to the Linker Response file as a search library. The following is an example of how the PRTSAMPLIB import library would be added to the Printer Setup Dialog procedure's Linker Response file, if an import library were used.

Version 1.3

```
prtsamp.obj
prtsamp.dll
nul.map
lcobol.lib+
os2.lib+
pmcob.lib+
pmbind.lib+
prtsamp.lib
prtsamp.def
/NOD /NOP
;
```

Version 2.0 (16-bit)

```
prtsamp.obj
prtsamp.dll
nul.map
lcobol.lib+
os2286.lib+
prtsamp
prtsamp.def
/NOD /NOP
;
```

An import library is created when the dynamic link library module is developed. The IMPLIB.EXE tool, part of the OS/2 Developer's Toolkit, builds the import library using the dynamic link library's Module Definition file as input. The IMPLIB.EXE tool requires, as input, the name to be assigned to the generated import library followed by the name of one or more Module Definition files. By allowing multiple Module Definition files to be specified, generalized import libraries covering more than one dynamic link library may be developed.

Here are the parameters required to run the IMPLIB.EXE tool, followed by the correct command to build the PRTSAMP.LIB import library for the Printer Setup Dialog procedure DLL, if one were used.

```
implib libname.lib    dllname1.def    [dllname2.def ...]
implib PRTSAMP.LIB   PRTSAMP.DEF
```

Building A Dynamic Link Library

With all of the assorted files created, the Printer Setup Dialog procedure and Printer Setup dialog template dynamic link libraries can be built. Here is a list of the steps required to build a dynamic link library module containing standard COBOL subroutines or functions. The Printer Setup Dialog procedure dynamic link library module is used in the examples.

- 1) Create the DLL's Linker Response file.

Version 1.3

```
prtsamp.obj
prtsamp.dll
nul.map
lcobol.lib+
os2.lib+
pmcob.lib+
pmbind.lib
prtsamp.def
/NOD /NOP
;
```

Version 2.0 (16-bit)

```
prtsamp.obj
prtsamp.dll
nul.map
lcobol.lib+
os2286.lib
prtsamp.def
/NOD /NOP
;
```

- 2) Create the DLL's Module Definition file.

```
LIBRARY PRTSAMP
DESCRIPTION 'OS/2 Print Dialog COBOL/2 DLL'
PROTMODE
CODE LOADONCALL
DATA SHARED
STACKSIZE 4096
HEAPSIZE 1024
EXPORTS PRTSAMPL @1
```

- 3) Create the source code for the dynamic link library subroutine or function.
- 4) Compile and link the DLL's source code.

```
@echo off  
Rem *** Compile and link a .dll program  
Cobol %1.cbl /CONFIRM /MF"6" /LITLINK;  
If errorlevel 1 goto end  
link @%1.l  
:end
```

Resource-Only Dynamic Link Libraries

Dynamic link libraries usually contain subroutines or functions, but they may also contain resource data. In fact, a dynamically linked module may contain only resource data and be completely free of usable code. I say usable code because a Resource file must be bound to an executable (.EXE or .DLL) file before it can be accessed. Thus, it is impossible to build a dynamically linked module containing only resource data. To satisfy the requirement for binding to an executable, a dummy program must be created to use as the nucleus of the resource bind process. This dummy program is referred to as the Resource Stub program.

The Resource Stub Program

The Resource Stub program is used only to facilitate the binding of the resource file and is never executed. As a result, it should contain no functions and should be kept as small as possible. The stub program can be written using any language, but is usually written using either the C or Macro Assembler languages for tighter control over the size and contents of the generated object file. COBOL could be used to compile a stub program, but the run-time support requirements make even a null COBOL program too large to use as a stub program.

A single stub program will work for every dynamically linked resource module, so the program only needs to be compiled once and the RCSTUB.OBJ file given to programmers responsible for generating the resource only DLLs. Here is the source code RCSTUB.C used to generate the C language stub program, RCSTUB.OBJ:

```

int_acrtused=1;

void far pascal RCSTUB()
{
}

```

When linking a resource only DLL, the same form of response file is used, but with some variations. The input object file on line 1 is the stub file, RCSTUB.OBJ. The output file on line 2 is a dynamic link library and requires the file extension .dll. Line 3 specifies that no Linker map is required and that only specified libraries should be searched. Line 4, the normal list of search libraries, is not required, as there are no external references to resolve in a stub program. Be sure to code a comma in place of the library list to indicate a null entry. Line 5 specifies the Module Definition file to use when linking a resource DLL. Following the Linker responses, the Linker directive /NOD is specified to suppress the searching of default libraries named in the object files. The following is the Linker Response file used to build the Printer Setup dialog resource template DLL:

Line 1	rcstub.obj
Line 2	prtdlgbox.dll
Line 3	nul.map
Line 4	,
Line 5	prtdlgbox.def
Line 6	/NOD
	;

A separate Module Definition file is required to link a resource only DLL. This file is essentially the same as the Printer Setup Dialog Module Definition file. The definition file contains the LIBRARY statement causing the Linker to build a dynamic link library module. The only other unique statement is the DATA statement. The NONE parameter does not indicate that no data segments are present, only that no automatic data segments are present.

The following is the Module Definition file for the Printer Setup resource template.

```

LIBRARY PRTDLGBX
DESCRIPTION 'OS/2 Print Dialog Resource'
PROTMODE
DATA NONE

```

Here is a list of the steps required to build a dynamic link library module containing only resource data. The Printer Setup dialog template dynamic link library module is used in the examples.

- 1) Create the resource data using an ASCII editor or the dialog template using the Dialog Editor. Place the dialog and any necessary header files into a Resource Script file.
- 2) Compile the Resource Script file into the Resource file.

```
rc -r prtdlgbx.rc
```

- 3) Compile the RCSTUB program. If you are not using the C language, then you must replace this C compiler command line with the correct command line for the compiler or assembler that you are using.

```
cl /c /Gs rcstub.c
```

- 4) Create the DLL's Linker Response file.

```
rcstub.obj  
prtdlgbx.dll  
nul.map  
'  
prtdlgbox.def  
/NOD  
;
```

- 5) Create the DLL's Module Definition file.

```
LIBRARY PRTDLGBX  
DESCRIPTION 'OS/2 Print Dialog Resource'  
PROTMODE  
DATA NONE
```

- 6) Link the Resource Stub file into a dynamic link library module.

```
Link @prtdlgbx.l
```

- 7) Bind the Resource file to the DLL executable.

```
rc prtdlgbx.res prtdlgbx.dll
```

The Printer Setup Dialog Procedure

I am not going to include specific information on writing the Printer Setup Dialog procedure. With two exceptions that I will discuss, there are no PM calls within this procedure that have not been used and explained in the previous chapters. The complete Printer Setup Dialog procedure is shown in Appendix A. I am, however, including information on the function of the dialog and how the dialog is implemented.

The Printer Setup Dialog procedure parses the OS2SYS.INI file and builds arrays containing all the printer queues and all the printer drivers available to the current system. This information is displayed to the user in two list boxes. The top list box shows all the printer queues defined for this system. The bottom list box shows all the printer drivers associated with the currently selected printer queue. The user is first asked to select the printer queue and then a printer driver from those associated with the selected printer queue. This printer queue and driver combination will be used to print the currently displayed PM call in the following chapter. Additionally, the user may specify the number of copies of the call to be printed. See Figures 16-3 and 16-4 for a picture and structure of this dialog.

When the user selects one of the push buttons to end the dialog, the PRTSAMPL function dismisses the dialog and returns the following information to the sample program.

DlgReturn

- The resource ID of the selected push button.

MPPrtCopies

- The number of copies to print.

MPQueueName

- The print queue selected by the user.

MPDriverName

- The Printer driver selected by the user.

MPDdrvName

- The left half of the driver name if a compound driver name was selected, otherwise the entire driver name.

MPPrtName

- The right half of the driver name if a compound driver name was selected.

A word of warning about this dialog. As a sample routine, the dialog does not account for the practical limits imposed upon production programs. Be aware that the arrays established to hold the printer queues and print drivers have a limit of five entries each. You may wish to change this limit to match your installation's norms.

The Printer Setup Dialog procedure is a standard COBOL program. There is no difference between the code structure of this dialog procedure and code structure of the dialog procedures contained within the main program. This is an important point. Many people perceive that dynamic link library functions must be specially written to function correctly. This is not true. Subroutines written for use in dynamic link libraries are

written exactly the same as any other COBOL PM subroutine. These subroutines have all of the standard COBOL divisions, storage areas and sections. In theory, the decision to make a subroutine a member of a dynamic link library can occur after the routine is written. In practice, however, significant planning is usually required before writing a dynamic link library subroutine, not because the code is different, but as a reentrant, shared procedure complex data handling may be involved.

The PRTSAMPL subroutine contains more than just the dialog. Entry into the subroutine is not directly into the dialog procedure but into a setup routine that saves the values passed to the subroutine, establishes addressability to the data pass area and loads the resource dynamic link library module, then invokes the dialog using the WinDlgBox call. Following the dismissal of the dialog, this routine sets the return code and exits the subroutine.

This chapter's sample program concentrates on dynamic linking and does not implement printing. Sufficient code has been added only to support dynamic link libraries and the Printer Setup dialog. Printer information selected during this dialog is ignored. The following chapter adds the code required to implement printing.

Coding The DosLoadModule Call

The module's setup routine contains a DosLoadModule call, one of the two new OS/2 calls added to the sample program in this chapter.

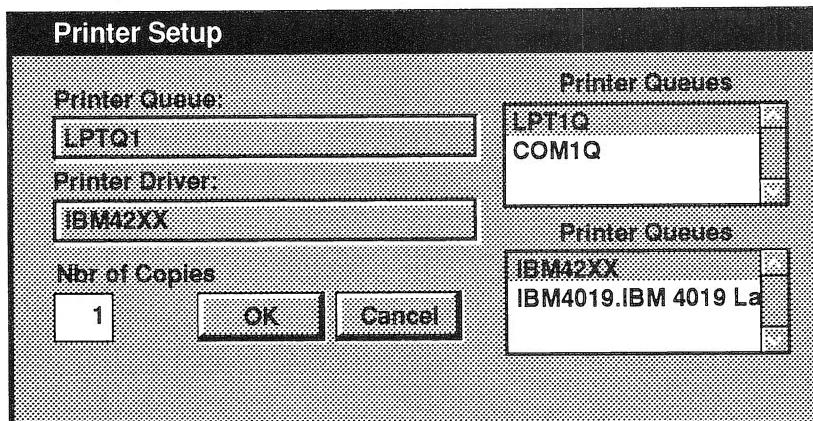


Figure 16-3 The Printer Setup dialog (Version 2.0)

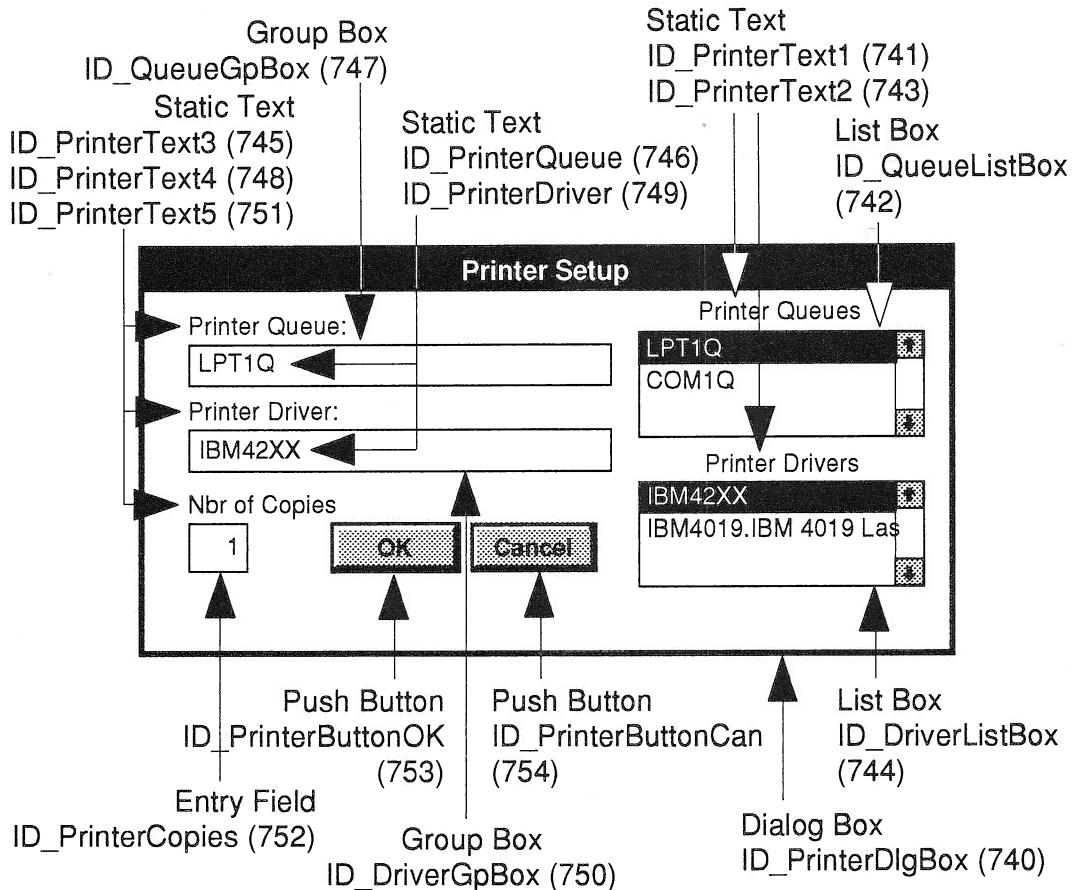
When a module is marked as LOADONCALL, it is automatically loaded when it is called, as is the Printer Setup Dialog procedure. However, when a module is marked as LOADONCALL but is never directly called, as with the Printer Setup dialog template, it must be loaded into memory by the application before its resources can be used. The DosLoadModule call finds the specified DLL and loads it into memory. The call returns the handle of the module, required to locate specific procedure addresses and to free the module after its use. If the referenced module is already in memory when the DosLoadModule call is made, module loading is suppressed and the handle returned.

To code the DosLoadModule call, place a pointer to a buffer large enough to hold the fully qualified drive, path and DLL module name in parameter one. The fully qualified module name will be returned if an error occurs during the DosLoadModule call. Parameter two is a short integer variable holding the length of the buffer pointed to by parameter one. Parameter three is a pointer to a null-terminated ASCII string holding the module name. If the module is stored in a subdirectory within the library search path, then only the module name need be specified. If the module is not stored in a directory within the library search path, then the fully qualified drive, path and module name must be specified. The Printer Setup dialog has the DLL module defined within Working-Storage as resident within a specific subdirectory, depending upon the version of OS/2. You may wish to change this definition. Parameter four is an unsigned short integer variable that will contain the module's handle upon return from this call. Note that unlike the long integer variables used to store other handles, parameter four is a short integer. The last parameter is the variable that will receive the call's return code. Here is the DosLoadModule call used to load the Printer Setup dialog template:

```
001870      Call OS2API 'DosLoadModule' using
001880              by reference ProfileString
001890              by value     200 size 2
001900              by reference DLLName
001910              by reference hwndDLL
001920              returning    ReturnData
```

Coding The DosFreeModule Call

DosFreeModule, the second of the two new OS/2 calls used in the Printer Setup Dialog, frees the DLL containing the dialog's resource template when the dialog is complete. Whenever a DLL that was loaded using the DosLoadModule is no longer required, it should be released so that its memory may be recovered by OS/2. The DosFreeModule call doesn't actually release the DLL, it only notifies OS/2 that the DLL is no longer required by the thread. OS/2 then decrements the DLL's use counter by 1. Only when



**Figure 16-4 The Printer Setup dialog box resources
(Figure shows Version 1.3 dialog box)**

the DLL's use counter reaches 0, indicating that all threads referencing the DLL no longer require it, will the DLL actually be released by OS/2 and the memory reclaimed.

To code the DosFreeModule, place the variable holding the handle of the module in parameter one. Parameter two is the variable that will receive the call's return code. Here is the DosFreeModule call used in the Printer Setup Dialog to free the reference to the dialog template:

003530
003540
003550

Call OS2API 'DosFreeModule' using
by value hwndDLL
returning ReturnData

Modifying The Sample Program

To ensure that Print menu entry can only be selected when the Source window is displaying a PM call, the Print menu entry is disabled when the program is started and remains disabled until the Source window is displayed. The last call in the *MI-Source* command routine issues the WinSendMsg call, sending an MM-SETITEMATTR message to the menu to enable the Print entry.

To ensure that the Print menu entry cannot be selected after the Source window is closed, the last entry in the *MI-Closesw* routine issues another WinSendMsg call, sending a MM-SETITEMATTR message to the menu to disable the Print entry.

When the MI-Print command is received, indicating that the user wishes to print the currently displayed *PM call*, the *MI-Print* command routine loads the two parameters that are to be passed to the Printer Setup Dialog procedure, then calls the external function PRTSAMPL. The first parameter passed is the handle of the client window, required for the WinDlgBox call made within the PRTSAMPL function. The second parameter passed is a pointer to the dialog data pass area that will contain the user-selected information at the completion of the dialog. Remember, MsgParm2 cannot be set to the address of the data area; only a variable declared as a pointer can be set to an address. So, the redefined MsgParm2, Msg2Pointer is used to hold the pointer to the dialog data passing area.

The actual call to the DLL subroutine PRTSAMPL is coded exactly like all other PM call. Do not be confused by my using the two standard message parameters for this call. This call does not generate a standard PM message, but is a standard COBOL subroutine call. This call treats MsgParm1 and MsgParm2 as two long integer variables, nothing more.

To code this call, place the long integer variable holding the handle of the Main window's Client window in parameter one. Code parameter two as a long integer variable containing the pointer to the dialog data pass area. The last parameter specifies the variable that will receive the user-selected push button when the call completes. Here is the PRTSAMPL call used in the sample program:

```
009960      Move hwndClient to MsgParm1
009970      Set Msg2Pointer to address of PMPrint
009980      Call OS2API 'PrtSampl' using
009990          by value    MsgParm1
010000          by value    MsgParm2
010010          returning DlgReturn
```

Chapter 17

Printing

Before beginning the discussion of OS/2 printing, a brief word about printing differences between OS/2 Version 1.3 and OS/2 Version 2.0. With the implementation of Version 2.0's object-oriented Workplace Shell, the OS/2 entity called the Print Manager, disappeared. In its place, Version 2.0 displays separate printer objects for each attached printer. While the Print Manager disappeared, the functions of the Print Manager have not, they have simply been shifted to the individual printer objects. For example, instead of opening the Print Manager to see a list of all jobs waiting to print, Version 2.0 users open the printer objects to see a list of jobs waiting to print on that specific printer. Or, instead of opening the Print Manager to add a printer driver, the Version 2.0 user open the settings of the specific printer object that will receive the new printer device driver.

In this chapter, I refer to the Print Manager several times. If you are using OS/2 Version 2.0, please associate the Print Manager information with the printer objects. All of the information is still valid, just under different names.

OS/2 printing is a complex process. But for every complex process OS/2 usually offers significant function and this is certainly true for printing. The much maligned Print Manager offers programmers a range of functions, from printer device independence to font management, each designed to assist your program in producing top-quality printed output. The key to quality OS/2 printing is understanding exactly what the Print Manager can and cannot do, and how to make it provide the printing services that your output requires.

To better understand the capabilities of the Print Manager and how it can help to add full-function, sophisticated printing to your applications, I have included a short discussion of the major Print Manager components, and the data streams and data flows that it supports. This is not a complete dissertation on OS/2 printing, rather more of a quick overview of some of the key points that you need to understand. If you are interested in a more detailed discussion of OS/2 printing, I recommend the book, *OS/2*

Version 1.3 Volume 2: Print Subsystem (GG24-3631), written by the IBM International Technical Support Center in Boca Raton, Florida. It is available through an IBM field representative. This book is a complete reference for the installation, support and use of the OS/2 1.3 Print Subsystem.

The OS/2 Printer Subsystem

Print Manager is a collective term for the multiple components that provide OS/2 printing. The Print Manager is composed of four modules: the user interface dialogs, the Queue Processor (often called the Spool Processor or the Spooler), the printer device drivers and the Kernel device drivers. Collectively these four components combine to provide printer device independence, spooling, data stream manipulation and physical printer control.

User Interface Dialogs

The user interface dialogs cover all of the dialogs that allow the user to install, modify and run the Print Manager. These dialogs allow the user to view and control the jobs waiting to be printed, view details about each of the jobs, install and define printer queues, logical printers and printer device drivers, associate logical printers with printer queues, define printer and job properties, and retrieve help information about the Print Manager.

The Queue Processor (Spooler)

The Queue Processor, often referred to as the Spooler, acquires the intercepted print data from the printer device driver and writes it to the proper \SPOOL subdirectory. Then, when all the criteria for printing the job have been met, it passes the data back to the printer device driver, which performs the actual printing. The Queue processor also provides code page support as needed. Currently, there are two queue processors, the default printer and plotter processor, PMPRINT.QPR and an optional queue processor, PMPLLOT.QPR, that supports standard data stream reverse clipping. PMPRINT.QPR is installed as the default queue processor during OS/2 installation.

For every print job, the Queue Processor creates two spooler files. The first, with the file extension of .SHD, contains control and job properties information that combine to determine how the data is printed. The second, with the file extension of .SPL, contains the actual data to be printed. For raw data streams, the SPL file holds the data in the exact format that will be passed to the printer. For standard data streams, the .SPL file holds the data in the untranslated metafile format sent by the program.

The Printer Device Driver

The printer device driver is the key element in the Print Manager. It supplies printer-specific services to the Print Manager for the printer that it controls. This is why it is so important that the correct printer device driver be installed for each printer. As the only repository of specific printer information, it supplies all the hardware, media and property information about the printer in response to a program's information query. The wrong printer device driver, or worse the NULL printer device driver, can cause significant problems for a program's print routine. Printing a report, for example, based upon the available printer features retrieved from a printer device driver that does not represent the actual printer can present a problem by indicating a specific feature is present, when, in fact, it is not.

A printer device driver is divided into two logical sections based upon the duties it must perform. Part one is responsible for intercepting the print data sent from the program or OS/2 command line and passing it to the Queue Processor. When the data is to be printed, part two receives the data from the Queue Processor, converts it into the proper data format for the printer, sends the necessary printer setup commands together with commands to invoke features and functions requested by the program, writes the data to the kernel device driver, and handles any error conditions that may arise.

The Kernel Device Driver

Physical control over the printers is provided by the kernel device driver controlling the port attaching the printer. Clearly, these drivers are not used exclusively for printing and they are not technically a part of the Print Manager. But they do provide a critical printing service and thus should be considered part of the Print Manager for any discussion of OS/2 printing.

While the printer device driver provides logical printer control, the kernel device driver provides physical control over the port and the data and control lines that attach the printer. Here are the various kernel device drivers used for printing under OS/2.

Version 1.3	Parallel port	Version 2.0
BASEDD02.SYS	Micro Channel	PRINT02.SYS
BASEDD01.SYS	Non-Micro Channel	PRINT01.SYS
Serial Port		
COMM02.SYS	Micro Channel	COM.SYS
COMM01.SYS	Non-Micro Channel	COM.SYS

Data Flow

There are three different routes that print data can follow between the program that creates it and the printer that prints it. Which route is taken, combined with the type of data stream transmitted, largely determines what services the Print Manager can and will provide.

Basic Printing

Basic printing is the transmission of a printer data stream directly from the program to the kernel device driver controlling the port attaching the target printer. By bypassing the Queue Processor entirely, none of the Print Manager's normal services are available, with the exception of data spooling for those printers attached to a parallel port. Basic printing requires the transmission of a raw data stream; that is, a data stream that contains data in the exact format required by the printer plus all of the printer control codes necessary to set up and control the printer. Basic printing is primarily the province of DOS programs, early OS/2 programs ported from DOS and the PRINT and COPY OS/2 commands. Basic printing includes all COBOL programs that print using the OPEN, WRITE and CLOSE statements. Because basic printing is supported using standard COBOL statements rather than PM calls, I will not be discussing basic printing in this book. See Figure 17-1 for the basic printing flow.

Queued Printing

While queued printing is the most difficult of the OS/2 printing data flows to implement, it offers the most options for advanced printing. By routing the print data through a queue rather than directly to a port, a program can invoke most of the services provided by the Print Manager. In combination with a standard printer data stream, queued printing supports printer device and data independence. That means the ability exists to print on any attached printer without altering the data stream, and the ability to implement advanced printer features outside of the program.

Queued printing derives its name from the fact that the program passes the data to a printer queue, not to the printer port, as is the case with basic printing. Rather than using the COBOL OPEN, WRITE and CLOSE statements, the program uses PM calls to open the desired printer queue, pass the output directly to the queue, then close the queue. The PM calls that must be used depend upon the type of data to be printed. If a printer independent standard data stream is to be built, then the data must be passed from a presentation space to the queue utilizing an OS/2 component called a device context. If a printer dependent raw data stream is to be built, then the presentation space can be bypassed and the data sent directly to the queue using the PM Spooler calls. See Figure 17-2 for the queued printing data flow.

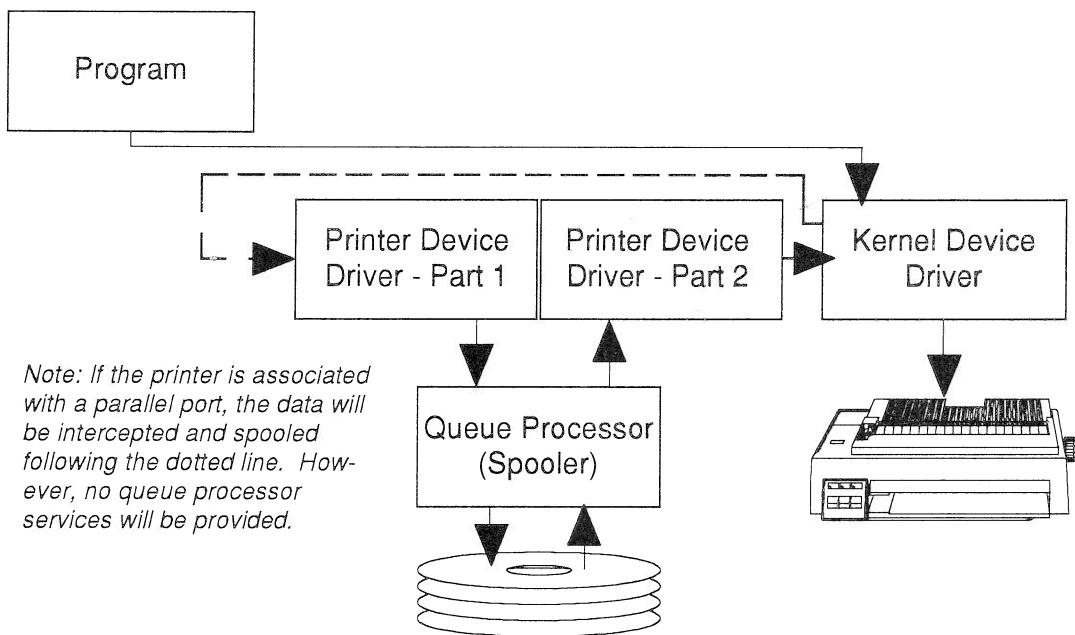


Figure 17-1 Basic printing data flow

Direct Printing

Direct printing utilizes all the features of queued printing except spooling. Think of direct printing as queued printing with the Print Manager turned off. Direct printing bypasses the spooler and sends the print output directly to the printer device driver. But unlike basic printing, direct printing can utilize a standard data stream to invoke all the functions of a printer device driver.

Direct printing should be used with great caution, as it can have a negative effect on programs running in a multitasking environment. Once printing begins, programs utilizing direct printing must wait until the printing is complete before they can continue processing; where multiple programs are attempting to print simultaneously, each must wait its turn before printing can begin. Since direct printing has the same program requirements as queued printing, there is little reason for a program to use direct printing. Direct printing is certainly not the proper printing scenario for the OS/2 environment. See Figure 17-3 for a diagram of direct printing.

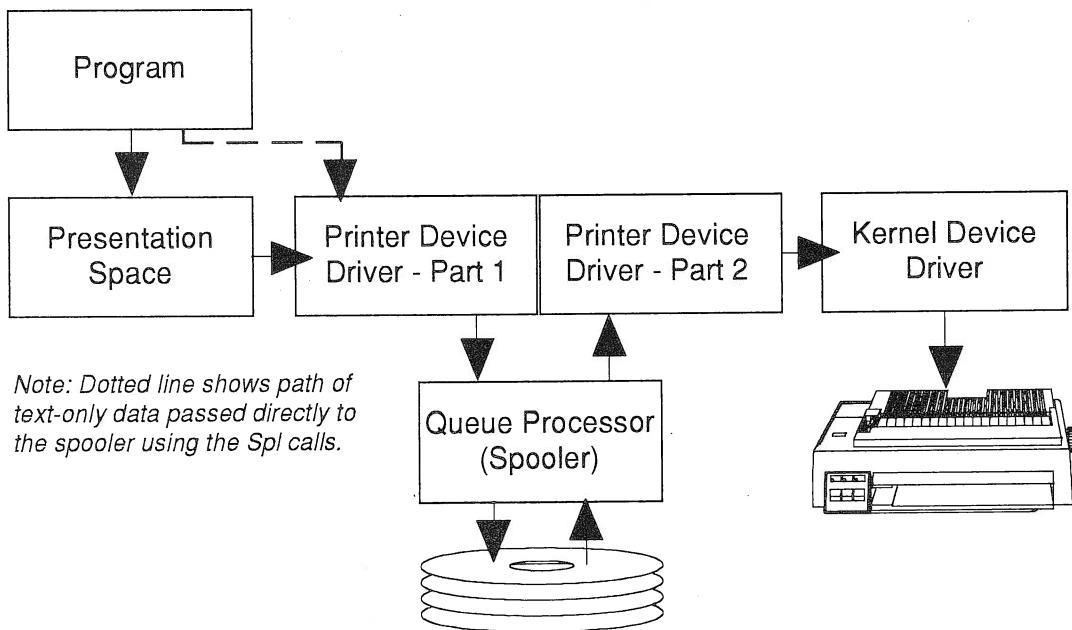


Figure 17-2 Queued printing data flow

Printer And Job Properties

When a program queries for information about a printer, much of what is returned comes from the printer properties. Print properties define the hardware options, fonts and paper handling ability of an attached printer. A printer's properties are established when the printer device driver for the printer is installed. Once set, printer properties should not be changed unless the printer's hardware, font or paper handling capabilities change. The printer device driver uses the properties to determine which printer features can be invoked. Printer properties include the following kind of entries:

- Default fonts
- Downloadable soft fonts
- Optional font cards and cassettes
- Forms available for the printer
- Installed memory
- Installed envelope feeder
- Number of paper trays
- Paper orientation
- Print densities

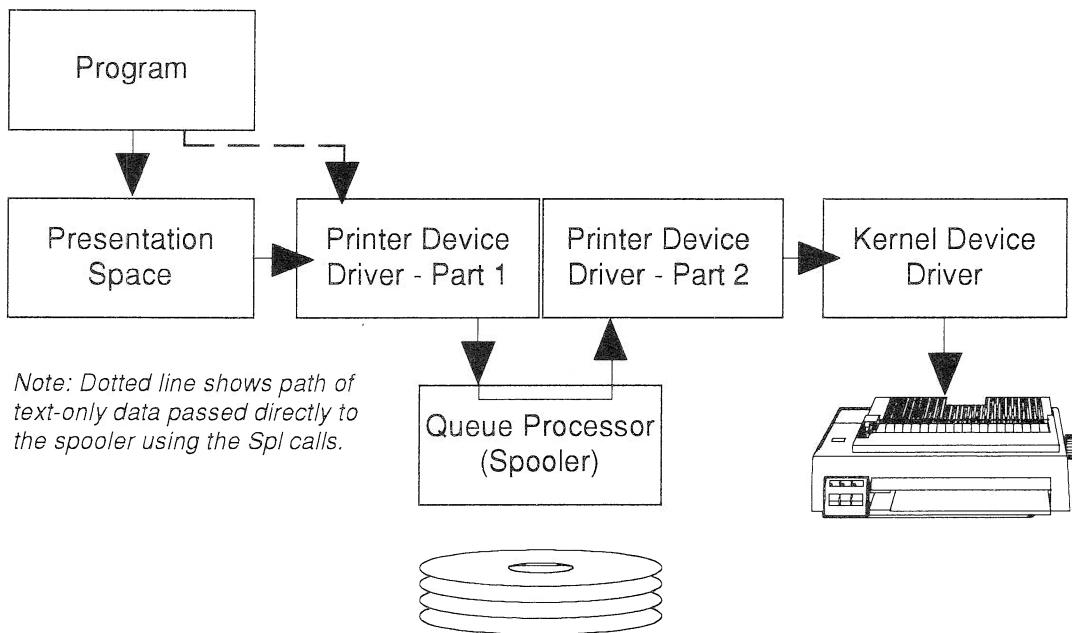


Figure 17-3 Direct printing data flow

Job properties are the subset of printer properties that may be manipulated by or for individual print jobs to alter their printer output. There are two sets of job properties.

Job properties are the properties assigned to the print data when the job is created. These properties are usually selected by the user through one or more job properties dialogs displayed during the print data build process.

Queue properties are the default job properties that will be assigned to jobs sent to the queue that do not contain related job properties. Queue properties will only be assigned if the job was received by the Queue Processor with no corresponding job property specified. For example, if a job is received by the Queue Processor with no paper orientation specified, then the paper orientation specified in the queue properties will be assigned. Queue properties allow raw data streams to manipulate some printer features by sending the data stream to the printer queue with the desired properties defined.

While job properties and queue properties are identical for a given printer, it is important to understand the order of their importance to the printer device driver. Job properties will always take precedence over any other properties. If no job property for a given feature or function is specified, then the queue property for that feature or function will be assigned. If no queue property for the feature or function is defined, then the default printer property will be assigned.

The Printer Data Stream

There are two types of data streams that may be sent to an OS/2 printer. In conjunction with the data flow path, the type of data stream largely determines the kind of support that the Print Manager will provide to a program. You must have a clear understanding of the support the Print Manager provides for each type of data stream, so that you know exactly what your program must provide and what services you can expect from the Print Manager.

Raw Data Streams (PM_Q_RAW)

This is the basic printer data stream and is identical to the printer data streams used within the DOS environment. A raw data stream is defined as a printer-specific data stream ready to be passed to the printer. A raw data stream must contain not only the text and graphics to be printed in the correct format for the target printer, but all the necessary control codes for printer setup, font and feature selection, paper management and printer shutdown. The Queue Processor and printer device driver will not examine or alter the contents of a raw data stream. If you will pardon the pun, for raw data streams what the program produces is what the printer gets.

Standard Data Streams (PM_Q_STD)

A standard data stream is a device independent data stream. It is composed of graphical drawing commands that define how to recreate the image of each page built by the program, not the actual data to be printed, as in a raw data stream. Being composed only of drawing commands, a standard data stream contains no printer-specific information and can be converted into a raw data stream for any printer or plotter by the appropriate device driver. The conversion from a standard data stream into a printer-specific or plotter-specific raw data stream occurs during the data's second pass through the device driver.

The independence of a standard data stream offers several advantages to the program creating the data.

- The print data can be directed to any attached printer.
- Printer data can be directed to a plotted and vice versa.
- Printed graphics are easier to include in the data stream and need to be created only once for display and printing.

- The program does not need to understand how to control the printer that does the printing.
- Unique printer features are specified generically, then invoked by the device driver during conversion.

Building A Standard Data Stream

There are just too many variables involved in the generation, placement and manipulation of data within the presentation space to set down specific rules that can always be followed when creating a standard data stream. The amount of user control over the output adds further to the complexity of standard data stream generation. If, for example, the program allows the user to select the font, the steps to build the output will be different than if the program chooses the font. So, what I have attempted to do is outline the major steps you must perform to construct and pass a standard data stream to the correct printer queue and printer device driver. I have included only the major PM calls. Obviously, there is much more code to complete the building of a data stream then just these calls. Consider this list as the minimum functions that you must perform. Here are the basic steps that a program must perform to print a standard data stream:

1) Query for the printer queue and driver.

Before any printing can take place, the program must determine the correct printer queue and printer device driver to use. This decision should be made by the user. Most OS/2 systems have access to too many printers to allow the program to decide which printer to use. To determine the correct queue and driver, the program must enumerate all the printer queues, including printer queues defined for remote LANs and their associated printer device drivers found in the OS2SYS.INI file. The printer queues and related device drivers are enumerated using the PrfQueryProfileString call with a PM_SPOOLER_PRINTER application name and a NULL key name. With the queue and device driver information displayed, the user is asked to first select a printer queue, then a printer device driver from those associated with the selected print queue. The code to accomplish this is contained in the sample program's Printer Setup dialog procedure. Being a self-contained DLL, this dialog procedure should fit nicely into any PM program. Please review this code in Appendix A.

2) Obtain the printer-specific driver data.

Following the selection of the printer queue and printer device driver, the program must obtain the printer specific information, including the properties, for the selected printer

by retrieving the associated printer device driver's Device-data structure. This data structure is obtained using the DevPostDevModes call. This call must be issued twice. First, with the pointer to the driver data save area set to null to retrieve the size of the Driver-data structure. The size of this structure can range from several hundred bytes to as large as 1K, depending upon the capabilities of the printer selected. With the size of the required save area available, the program must allocate memory to hold the returned structure using the DosAllocSeg call for 16-bit programs, and the DosAllocMem call for 32-bit programs. This call dynamically allocates space outside of the program's data area to hold this data structure. Finally, the DevPostDevModes call is issued a second time with parameter two containing the address of the allocated memory. This time, the call returns a structure of driver-specific data for the specified printer. The code to obtain the driver data is in the sample program starting at line 010430.

You have an option when issuing the DevPostDevModes call that I must alert you to because it can add so much flexibility to any printing routine. Parameter six of the DevPostDevModes call specifies the user's dialog options for job properties. If you specify DPDM-POSTJOBPROP, as I have done in the sample program, the printer device driver will pop up the printer's job properties dialog and allow the user to make changes that will apply only to the current print job. The updated information is then returned as the Device-data structure. The DPDM-POSTJOBPROP option gives the user added printer flexibility without adding a single line of code to the program. You can alternately specify DPDM-CHANGEPROP, which displays the printer properties dialog, making any user changes to the job properties permanent. Finally, if you do not want the user to see the dialog, you can specify DPDM-QUERYJOBPROP to skip the dialog display and return the Driver-data structure.

3) Create a device context for the printer.

The program must now tell PM the type of output device that will receive this data. The transfer of data from a presentation space to a physical device is accomplished using a device context. There are device contexts for display screens and printers and plotters. So, a printer device context must be created for printer output using the DevOpenDC call. During this call, the type of data flow is set, OD-QUEUED for queued printing or OD-DIRECT for direct printing. Information unique to the target printer device driver is made available to the device context by passing a pointer to the Driver-data structure as part of the DevOpenDC call.

4) Obtain the size of the printer media.

To create a presentation space with the exact size of the printer media, the program must obtain the size of the printer media using the DevQueryCaps call. Parameter one of this call identifies the printer by specifying the device context. Parameter four returns an

array containing 39 physical properties of the target printer, among them the media width and height specified in pels.

5) Create a presentation space for the output.

With the device context established and the size of the media available, the program must create a presentation space of the same size as the printer media using the GpiCreatePS call. By creating the presentation space the same size as the media, data positioned correctly in the presentation space will be positioned correctly on the paper. Parameter two associates the printer device context with the presentation space and parameter three sets the size of the presentation space based upon the printer media. Don't forget to define the presentation space in pels, so that the printer and presentation space will be using the same scale. Specify pels by coding parameter four as PU-PELS.

6) Determine printer support.

If there are specific printer escape sequences that the program must use, the program can query the printer device driver at this point to determine if they are supported. By issuing the DevEscape call with parameter two set to DEVESC-QUERYESCSUPPORT, the program will be told if the printer escape code specified in parameter four is supported. The predefined DevEscape codes for parameter two are listed in Appendix D.

7) Establish the font to be used.

By this point, the program knows the size of the printer media and the data to be printed. The last remaining function is to establish the font to be used. This can be complex or relatively simple, depending upon the extent of user involvement. If, as in the sample program, the program makes the font decision, then this is a fairly straightforward function. If the user is allowed to make the font decision, then this function will include a fairly complex dialog.

Regardless of who makes the font decision, here are the calls required. First, query for the available fonts, probably limiting the query to a single font face, Courier for monospaced type, any other font face for proportionally spaced type. This call must be issued twice, first with parameter four set to 0 to retrieve the number of fonts within the specified font face. Then, after obtaining memory to hold the font metrics, the GpiQueryFonts call is reissued to retrieve the actual metrics. With all the metrics available, the program must parse the metrics to find a usable font. When the desired font is found, the match number is saved to use when building the logical font. Here's a tip. If the font's match number is negative, then the font is a printer device font; that

is a font available either in the printer or as a downloadable soft font. If the match number is positive, then the font is a general public font and may not be available for the printer. If you request the use of a non-device font within a printer standard data stream, the printer device driver will convert the entire output to a bit mapped image in an attempt to match the requested font.

The match number and the font face name are placed into the Font Attribute table together with any additional font attributes (bold, italics, underline, etc.) and the logical font created using the GpiCreateLogFont call.

8) Start printing the document.

At this point, the program begins to output data to the printer queue using the DevEscape call with the call's code set to DEVESCAPE-STARTDOC. This call causes the .SHD file to be built and the .SPL file to be opened in the queue's spool subdirectory, and prepares the spooler to begin receiving the printer data.

9) Create the logical font.

Before the logical font can be created, the program must tell the presentation space, via the GpiSetCharMode call, whether an image or outline font will be used. The mode selected limits the amount of control that may be exercised by drawing commands to the flexibility offered by the selected image or outline type font.

Next, the program sets the size of the character box using the GpiSetCharBox call. The character box is a logical rectangle that determines the size and placement of each character on a line. For image fonts, the size of the characters is determined by the font and cannot be changed. Since image fonts cannot be sized, set the character box to the value in the *Fmetrs-Eminc* font metrics entry. For outline fonts, the size of the font is determined by the size of the character box. Remember, outline fonts are scalable and selecting the font face does not define the font's size. So the character box should be set to the desired font point size to cause PM to create the correctly sized logical font. Be sure to specify the point size in the same page-size units specified when the presentation space was created.

At this point, everything is in place to create the font using the GpiCreateLogFont call. This call uses the information set by the prior calls to build the correct font translation table within the presentation space. But building the font table is not enough. Only the current font is used when data is placed into the presentation space, so the newly created font must be made the current font using the GpiSetCharSet call. Now, everything is ready to draw the output to the presentation space. For more information on using fonts, see Chapter 12.

10) Drawing the output.

Text and graphical images may be placed into the presentation space using several different Gpi calls. The exact calls used are largely determined by the amount of control over the placement of individual characters and images that the program requires. As an example, here are just a few of the functions a program must perform when drawing data into the presentation space.

- The program must determine the number of characters to place on each line based upon the font size, its spacing attributes and the line justification.
- The program must determine the exact vertical location of each print line using the font's kerning value.
- The program must vary the font or font attributes when changes in the output are required.
- The program must correctly insert the required new line marker at the end of each line and an end-of-page marker at the end of each page.

The actual placement of the text is accomplished using the GpiSetCurrentPosition call to set the starting point for the text string and the GpiCharString call to draw the text string into the presentation space. Graphical images are not placed into a presentation space, but drawn using arc, circle, line, etc., drawing commands used to create the display of the same image.

At the end of each page, the program must issue a DevEscape call specifying the DEVESCAPE-NEWFRAME code to transfer the contents of the current page and begin a new page.

At the end of the output, the program issues a DevEscape call specifying the DEVESCAPE-ENDDOC code to end the document, close the spool and begin printing the data.

11) Destroy the allocated resources.

With printing complete, the program must return or destroy the resources that were created on its behalf. Of course, if they are to be used again, they can be retained, but as a general rule they should be returned to PM and recreated again when needed. Presentation spaces and logical fonts consume enormous amounts of memory. To dispose of the allocated resources, free any allocated memory using a DosFreeSeg call for each allocated segment (16-bit programs), and a DosFreeMem for each allocated memory

Building A Raw Data Stream

The steps required to build a raw data stream are very similar to those required to build a standard data stream when queued or direct printing is involved. The major difference between a standard data stream and a raw data stream is the assumption by the program of the functions performed in the presentation space and by the device context, when building a raw data stream. The sample program uses a raw data stream to print the PM call selected by the user.

Electing to send a raw data stream, the sample program eliminated the need for a presentation space by constructing the data stream within the program. This means that along with the text to be printed, any required printer control codes must be inserted into the data stream where they will be required by the printer. You will notice that in the sample program every line of text is ended with specific codes to perform a carriage return (x'0D') and line feed (x'0A'). These are specific control codes that the sample program assumes will be interpreted by the printer as a CR and LF. If the user directs this output to a printer that does not recognize these codes as a CR and LF, then the data will be printed incorrectly. Remember, the Queue Processor and the printer device driver will not examine or alter a raw data stream. Contrast this with a standard data stream that uses a generic control codes that any printer device driver can translate into the correct CR and LF codes used by the target printer.

Here are the basic steps that a program must perform to print a raw data stream:

1) Query for the printer queue and driver.

Before any printing can begin, the program must determine the correct printer queue and printer device driver to use. This query is identical to queue and device driver query used for a standard data stream. You may wish to review the first item under *Building a Standard Data Stream*. The process of determining which printer queue and printer device driver to use is contained in the sample program's Printer Setup dialog, PRTSAMPL.DLL. Please see Chapter 14 for initialization file processing and Appendix A for the Printer Setup dialog procedure code.

2) Obtain the printer driver data.

When building a raw data stream, the Driver-data structure is obtained exactly as it is for a standard data stream, using the DevPostDeviceModes call. Again, this call must be issued twice, first with the buffer pointer set to null to retrieve the size of the data structure, then a second time to actually acquire the structure.

Between the two calls, the program must ask OS/2 to allocate space for this structure using the DosAllocSeg call (16-bit program) and DosAllocMem call (32-bit program). This call allocates a segment of the size requested, pointed to by the address returned in parameter two of the call.

The DosAllocSeg call is coded by placing the size of the requested memory block in parameter one. The size of this memory must be at least as large as the size of the Driver-data structure returned by the prior DevPostDeviceModes call. Parameter two is a pointer to the variable that will receive the selector number. Because the actual buffer address must be defined as a selector number and an offset, the returned selector number is stored as part of the redefined DriverData variable. To position the data structure at the start of the segment, the offset value is set to 0 prior to the call. Remember, the selector and offset must be coded in the reverse order, with the offset followed by the selector number to comply with the Intel coding conventions. The third parameter contains flags that define the sharing characteristics of the allocated segment. For the sample program, the data segment is not shared, so no flags are set to true. The last parameter is the variable that will contain the call's return code. Here is the DosAllocSeg call used in the sample program preceded by the declaration of the segment address variable DriverData.

```

004410 05 DriverData usage is pointer.
004420 05 Redefines DriverData.
004430    10 AddrOffset          pic 9(4) comp-5.
004440    10 SelectorNbr      pic 9(4) comp-5.

010200            Move 0 to AddrOffset
010210            Call OS2API 'DosAllocSeg' using
010220                by value     UShortWork
010230                by reference SelectorNbr
010240                by value     0 size 2
010250                returning   ReturnData

```

Because the returned selector is the equivalent of an address, when it is passed to PM in the following DevPostDeviceModes call, the *by reference* phrase is not specified. Specifying *by reference* would pass the address of the variable holding the selector and offset, rather than the actual selector and offset values themselves.

The DevPostDeviceModes call is coded by placing the variable holding the handle of the Anchor-block in parameter one. Parameter two is coded as a long null integer when the size of the structure is to be returned, and with the variable holding the selector and offset of the allocated segment when the Driver-data structure is to be returned. Parameter three is the variable holding the printer device driver name. Parameter four is the variable holding the printer device type name. For printer device drivers with

compound names, those that include a period such as PScript.IBM 4019 v52_1 (39 Fonts), parameter three is the name to the left of the period and parameter four is the name to the right of the period. If the printer device driver does not have a compound name, such as IBM42XX, then parameter three contains the entire device driver name and parameter four is coded as a null. Parameter five is a pointer to the string containing the logical printer device name, for example PRINTER1, as established by the user during printer driver setup. If the call's option parameter is coded as DPDM-POSTJOBPROP, then parameter five is not used and can be coded as a null. Parameter six is coded as a long integer containing the call's option. For the sample program, this is coded as DPDM-POSTJOBPROP. The last parameter is the variable that will receive the call's return code.

Here is the DevPostDeviceModes call as used in the sample program to retrieve the Driver-data structure:

010290	Call OS2API 'DevPostDeviceModes' using
010300	by value hab
010310	by value DriverData
010320	by reference PMPDrvName
010330	by reference PMPPrtName
010340	by value LongNull
010350	by value DPDM-POSTJOBPROP
010360	returning ReturnData

Following the retrieval of the Driver-data structure, those DevOpenDataStructure parameters that will be varied for this print job must be updated. The sample program gives an example of how to update the DevOpenDataStructure. The sample program adds the name of the printer queue and printer device driver that are to be used, the type of printer data stream that will be passed and the number of copies to be printed. The comments field and Queue Processor parameters are not used by the sample program, but must be set to null as fields in the structure beyond these two are used. All fields that lie within the count specified in the call used to pass the DevOpenDataStructure to the Queue Processor, SplQmOpen for the sample program, must have a valid entry. Here is the code used to load the DevOpenDataStructure prior to it being passed to the Queue Processor as part of the SplQmOpen call.

010400	Set DeviceAddress to address of PMPQueueName
010410	Set DDRiverName to address of PMPDriverName
010420	Set DDataType to address of PrtDataType
010430	Set Comment to address of NullString
010440	Set DQueueName to address of NullString
010450	Set QueueParms to address of NbrPrtCopies
010460	Move PMPrtCopies to PrtCopies

At this point, you see one of the significant drawbacks to using a raw data stream. By eliminating the presentation space and device context, the program cannot issue the DevQueryCaps call to retrieve information about the target printer. Without the DevQueryCaps call, printer-specific information must be hardcoded into the program. This effectively eliminates the ability to direct print output to any attached printer, as the codes and data structure used in the program may not be compatible with the printer selected. The inability of prior operating systems to supply printer information to a program during execution is one of the reason applications have had to prepackage printer-specific information as Printer Function Tables. It is not uncommon for applications to contain forty or more printer function tables in an effort to support as many printers as possible.

The sample program avoids this entire problem by holding the width of the output to a size that almost every printer can accommodate, and limiting output to a single page. I did this because I wanted to concentrate on the PM calls required to perform printing, rather than worry about formatting the output for every possible printer you might have. But, this is not the real world environment and you will need to know your target printers, if you do not use the standard data stream output.

3) Start printing the document.

As with a standard data stream, output begins when a spool is opened and the initial spool files created. However, because there is no device context, the program must obtain the handle of the queue so the printer output can be passed directly to the Queue Processor. The handle of the queue is obtained using the SplQmOpen call. This call is coded by placing a pointer to a null-terminated token string in parameter one. This token identifies printer information stored in the initialization file or, if coded as an asterisk (*), indicates that no printer data is stored in the initialization files. If an asterisk is coded, the Queue Processor looks to the DevOpenDataStructure as the sole source of printer information. Always use the asterisk token and pass the data as part of the call. Since the DevOpenDataStructure is required, the initialization file is redundant in most cases. Parameter two is the count of the number of items in the DevOpenDataStructure. This count may be less than the number of entries in the data structure if fewer parameters are passed, but every parameter included in this count must be initialized, even if they are not used. Parameter three is a pointer to the DevOpenDataStructure, which holds both the printer selections and the Driver-data structure. The last parameter is the long integer to hold the returned queue handle.

Here is the SplQmOpen call used in the sample program along with the declaration of the token string:

```

002330 77 Token          pic x(2) value "*" & x"00".

010470                  Call OS2API 'SplQmOpen' using
010480                      by reference Token
010490                      by value      7 size 4
010500                      by reference DevOpenDatStruct
010510                      returning    hspl

```

With the handle of the queue now available, a spool must be created to hold the data. Similar in function to the standard data stream's DevEscape DEVESC-STARTDOC, the SplQmStartDoc call is issued. Like the DevEscape call, this Spl call causes the Queue Processor to open a spool, build the .SHD control file and open the .SPL file. The SplQmStartDoc call requires the handle of the queue as parameter one. Parameter two is a pointer to the document name to be used by the Print Manager. This is the name that the user will see when displaying print jobs held in the print queue. The last parameter is an unsigned short integer that will receive the call's return code. Here is the SplQmOpen call used in the sample program:

```

010540                  Call OS2API 'SplQmStartDoc' using
010550                      by value      hspl
010560                      by reference PrinterDocName
010570                      returning    ReturnData

```

4) Passing the output to the Queue Processor.

The actual lines of output are transferred using the SplQmWrite call. This call is coded by placing the handle of the queue in parameter one. Parameter two is a long integer containing the length of the data to be passed to the buffer. Null-termination characters cannot be used with printed output so a length must be passed. The length of the data may not exceed 65,535 characters. More than one print line may be passed with each SplQmWrite call. The limit of 64k means multiple lines are possible, but where multiple lines are used, the printer-specific new line control codes must be inserted into the data at the proper points to ensure correct line breaks. Parameter three is a pointer to the buffer containing the output to be transferred to the spool. The last parameter is a short unsigned integer to receive the call's return code. Here is the SplQmWrite call used in the sample program:

```

COBOL/PM Call = WinCreateStdWindow
-----
call OS2API 'WinCreateStdWindow' using
    by value      HWND-DESKTOP
    by value      ULONGNULL
    by reference  FCF-MAIN
    by reference  MainWndClass
    by reference  NullString
    by value      ULONGNULL
    by value      USHORTNULL
    by value      ID-MainWind
    by reference  hwndClient
    returning     hwndFrame

```

Figure 17-4 The sample program's printed output

```

013760      Call OS2API 'SplQmWrite' using
013770                  by value      hspl
013780                  by value      LongWork
013790                  by reference PrintLine
013800                  returning   ReturnData

```

When the last line of output has been sent to the spool, the end of the document must be marked. This is done using the SplQmEndDoc call, the equivalent of the standard data stream's DevEscape DEVESCAPE-ENDDOC. When the last of the output data is in the queue, the queue must be closed so that the data can be released to the printer device driver for printing. No printing will begin until the queue is closed by the application. The queue is closed using the SplQmClose call.

Both of these calls are coded using the same parameters. Parameter one is the variable holding the handle of the queue, and the second parameter is the variable that will receive the call's return call. For the SplQmEndDoc call, the return code is the job number assigned to the output by the Queue Processor. Code this variable as an unsigned short integer.

Here are the SplQmEndDoc and SplQmClose calls used in the sample program:

```

010670          Call OS2API 'SplQmEndDoc' using
010680                  by value      hspl
010690                  returning   ReturnData

```

```
010710          Call OS2API 'SplQmClose' using
010720                  by value hspl
010730                  returning ReturnData
```

5) Destroy the allocated resources

With no presentation space or logical font the only resources to free are allocated memory segments. As with the standard data stream, release all allocated segments using the DosFreeSeg call for 16-bit programs or the DosFreeMem call for 32-bit programs. To code the DosFreeSeg call, place the variable holding the selector number of the segment to be freed as parameter one, and the variable that will receive the call's return code as parameter two. Here is the DosFreeSeg call used in the sample program:

```
010750          Call OS2API 'DosFreeSeg' using
010760                  by value SelectorNbr
010770                  returning ReturnData
```

```
000010 IDENTIFICATION DIVISION.  
000020 PROGRAM-ID. COBOL-PM-SAMPLE.  
000030 DATE-COMPILED. 01-02-92.  
000040 AUTHOR. DAVID DILL.  
000050  
000060 ENVIRONMENT DIVISION.  
000070 CONFIGURATION SECTION.  
000080 SOURCE-COMPUTER. IBM-PERSONAL-SYSTEM-2.  
000090 OBJECT-COMPUTER. IBM-PERSONAL-SYSTEM-2.  
000100  
000110 SPECIAL-NAMES.  
000120      call-convention 3 is OS2API.  
000130  
000140 INPUT-OUTPUT SECTION.  
000150 FILE-CONTROL.  
000160      Select Source-Listing assign to UT-S-SourceList  
000170      Organization is line sequential  
000180      Access mode is sequential.  
000190 DATA DIVISION.  
000200 FILE SECTION.  
000210 FD Source-Listing  
000220      Record is varying in size from 8 to 60 characters  
000230      Recording mode is variable  
000240      Label record is standard.  
000250 01 SourceRecord.  
000260      05 Statement.  
000270          10 REC-ID           pic x(30).  
000280          10 Filler          pic x(30).  
000290  
000300 WORKING-STORAGE SECTION.  
000310*  
000320* Presentation Manager message structure  
000330*  
000340 01 QMSG.  
000350      05 QMSG-HWND          pic 9(9) comp-5.  
000360      05 QMSG-MSGID         pic 9(4) comp-5.  
000370      05 QMSG-PARAM1        pic 9(9) comp-5.  
000380      05 QMSG-PARAM2        pic 9(9) comp-5.  
000390      05 QMSG-TIME          pic 9(9) comp-5.  
000400      05 QMSG-POINT         pic 9(9) comp-5.  
000410          10 QMSG-X           pic 9(9) comp-5.  
000420          10 QMSG-Y           pic 9(9) comp-5.  
000430*  
000440* Window handles  
000450*  
000460 77 hab                 pic s9(9) comp-5.  
000470 77 hmq                 pic s9(9) comp-5.  
000480 77 hwndClient          pic s9(9) comp-5.  
000490 77 hwndFrame           pic s9(9) comp-5.  
000500 77 hwndPointer          pic s9(9) comp-5.  
000510 77 hwndLPointer         pic s9(9) comp-5.
```

```

000520 77 hwndDPointer          pic s9(9) comp-5.
000530 77 hwndSourceClient     pic s9(9) comp-5.
000540 77 hwndSource           pic s9(9) comp-5.
000550 77 hwndMenu             pic s9(9) comp-5.
000560 77 HWND-DESKTOP         pic s9(9) comp-5 value 1.
000570 77 HWND-TOP            pic s9(9) comp-5 value 3.
000580 77 hwndChildWindows     pic s9(9) comp-5.
000590 77 hwndChildQuery       pic s9(9) comp-5.
000600 77 hwndSWindow          pic s9(9) comp-5.
000610 77 hwndSWindowQuery     pic s9(9) comp-5.
000620 77 Hini-MyProfile       pic s9(9) comp-5.
000630 77 HINI-USERPROFILE     pic s9(9) comp-5 value -1.
000640 77 hspl                pic s9(9) comp-5.
000650*
000660* Window Classnames and titles
000670*
000680 77 MainWndClass         pic x(9) value "MainClas" & x"00".
000690 77 ChildWndClass        pic x(9) value "ChldClas" & x"00".
000700*
000710* Class Style Flag
000720*   CS-CLIPCHILDREN      0x20000000
000730*   CS-CLIPSIBLINGS        0x10000000
000740*   CS-CLIPPARENT         0x08000000
000750*
000760 77 CSClass              pic 9(9) comp-5 value h"20000000".
000770 77 CSChildClass         pic 9(9) comp-5 value h"38000000".
000780*
000790* Window Creation Flags
000800*   FCF-TITLEBAR         0x00000001
000810*   FCF-SYSCMENU          0x00000002
000820*   FCF-MENU              0x00000004
000830*   FCF-SIZEBORDER         0x00000008
000840*   FCF-MINBUTTON         0x00000010
000850*   FCF-MAXBUTTON         0x00000020
000860*   FCF-BORDER             0x00000020
000870*   FCF-TASKLIST          0x000000800
000880*   FCF-ICON               0x00004000
000890*   FCF-ACCELTABLE        0x00008000
000900*
000910 77 FCF-MAIN             pic 9(9) comp-5 value h"0000ca3f".
000920 77 FCF-CHILD             pic 9(9) comp-5 value h"0000820d".
000930*
000940* Set-Window-Position Flags
000950*   SWP-SIZE              0x00000001
000960*   SWP-MOVE              0x00000002
000970*   SWP-ZORDER             0x00000004
000980*   SWP-SHOW               0x00000008
000990*   SWP-ACTIVATE           0x000000080
001000*
001010 77 SWP-WINDOW            pic 9(4) comp-5 value h"0000008f".
001020 77 SWP-DIALOG             pic 9(4) comp-5 value h"00000006".

```

```
001030*
001040* System Query values
001050*
001060 77 SV-CXSIZEBORDER          pic 9(4) comp-5 value 4.
001070 77 SV-CXSCREEN             pic s9(4) comp-5 value 20.
001080 77 SV-CYSCREEN             pic s9(4) comp-5 value 21.
001090*
001100* PM Message IDs
001110*
001120 77 WM-COMMAND             pic 9(4) comp-5 value 32.
001130 77 WM-PAINT               pic 9(4) comp-5 value 35.
001140 77 WM-QUIT                pic 9(4) comp-5 value 42.
001150 77 WM-CONTROL              pic 9(4) comp-5 value 48.
001160 77 WM-INITDLG              pic 9(4) comp-5 value 59.
001170 77 WM-MOUSEMOVE            pic 9(4) comp-5 value 112.
001180*
001190* Frame Control IDs
001200*
001210 77 FID-MENU               pic 9(4) comp-5 value 32773.
001220*
001230* Menu Control Messages
001240*
001250 77 MM-SETITEMATTR          pic 9(4) comp-5 value 402.
001260 77 MIA-DISABLED            pic 9(4) comp-5 value h"4000".
001270*
001280* List Box Commands and Messages
001290*
001300 77 LIT-END                pic s9(9) comp-5 value -1.
001310 77 LN-SELECT               pic 9(4) comp-5 value 1.
001320 77 LN-ENTER                pic 9(4) comp-5 value 5.
001330 77 LM-INSERTITEM            pic 9(4) comp-5 value 353.
001340 77 LM-SELECTITEM            pic 9(4) comp-5 value 356.
001350 77 LM-QUERYSELECTION        pic 9(4) comp-5 value 357.
001360 77 LM-DELETEALL             pic 9(4) comp-5 value 366.
001370*
001380* Button Commands and Messages
001390*
001400 77 BN-CLICKED              pic 9(4) comp-5 value 1.
001410 77 BM-QUERYCHECK            pic 9(4) comp-5 value 292.
001420 77 BM-SETCHECK              pic 9(4) comp-5 value 293.
001430*
001440* Entry Field Commands and Messages
001450*
001460 77 EM-QUERYCHANGED           pic 9(4) comp-5 value 320.
001470 77 EM-SETTEXTLIMIT            pic 9(4) comp-5 value 323.
001480*
001490* Resource File Definitions
001500*
001510 77 ID-MainWind              pic 9(4) comp-5 value 10.
001520 77 MI-PRINT                 pic 9(4) comp-5 value 115.
001530 77 MI-Exit                  pic 9(4) comp-5 value 116.
```

```
001540 77 MI-Wedit          pic 9(4) comp-5 value 123.
001550 77 MI-Source         pic 9(4) comp-5 value 132.
001560 77 MI-About          pic 9(4) comp-5 value 162.
001570 77 ID-SourceWind     pic 9(4) comp-5 value 20.
001580 77 MI-Closesw        pic 9(4) comp-5 value 211.
001590 77 IDS-PMCall11      pic 9(4) comp-5 value 500.
001600 77 ID-TermDlgBox    pic 9(4) comp-5 value 700.
001610 77 ID-TermOK         pic 9(4) comp-5 value 701.
001620 77 ID-PMCall1DlgBox  pic 9(4) comp-5 value 705.
001630 77 ID-PMCall1ListBox pic 9(4) comp-5 value 706.
001640 77 ID-PMCall1ButtonCan pic 9(4) comp-5 value 707.
001650 77 ID-PMCall1ButtonOK pic 9(9) comp-5 value 708.
001660 77 ID-PMCall1Chosen  pic 9(4) comp-5 value 710.
001670 77 ID-BN-White       pic 9(4) comp-5 value 700.
001680 77 ID-BN-Blue        pic 9(4) comp-5 value 701.
001690 77 ID-BN-DK-Gray     pic 9(4) comp-5 value 708.
001700 77 ID-BN-DK-Blue     pic 9(4) comp-5 value 709.
001710 77 ID-BN-DK-Red      pic 9(4) comp-5 value 710.
001720 77 ID-BN-DK-Green    pic 9(4) comp-5 value 712.
001730 77 ID-BN-Brown       pic 9(4) comp-5 value 714.
001740 77 ID-BN-LT-Gray     pic 9(4) comp-5 value 715.
001750 77 ID-WinEditDlgBox  pic 9(4) comp-5 value 716.
001760 77 ID-CB-MainWindow  pic 9(4) comp-5 value 719.
001770 77 ID-WindowName     pic 9(4) comp-5 value 721.
001780 77 ID-WinDlgButtonCan pic 9(4) comp-5 value 724.
001790 77 ID-PrinterButtonCan pic 9(4) comp-5 value 754.
001800 77 IDS-MsgBoxOne     pic 9(4) comp-5 value 801.
001810 77 IDS-MsgBoxTwo     pic 9(4) comp-5 value 802.
001820 77 ID-Handptr        pic 9(4) comp-5 value 901.
001830 77 ID-LHandptr       pic 9(4) comp-5 value 902.
001840 77 ID-DHandptr       pic 9(4) comp-5 value 903.
001850*
001860* Message Box Declarations
001870*
001880* MB-OK             0x0000
001890* MB-INFORMATION    0x0030
001900* MB-CRITICAL       0x0040
001910*
001920 77 MB-INFO-OK       pic 9(4) comp-5 value h"0030".
001930 77 MB-CRITICAL      pic 9(4) comp-5 value h"0040".
001940*
001950* Window Colors
001960*
001970 77 Clr-White        pic s9(2) comp-5 value -2.
001980 77 Clr-Black         pic s9(2) comp-5 value -1.
001990 77 Clr-Blue          pic s9(2) comp-5 value 1.
002000 77 Clr-Yellow        pic s9(2) comp-5 value 6.
002010 77 Clr-PaleGray      pic s9(2) comp-5 value 15.
002020*
002030* Font Enumeration Options
002040*
```

```

002050 77 QF-PUBLIC
002060*
002070* Window words offset value
002080*
002090 77 QWL-OldProc
002100 77 QWL-hwndWindow
002110*
002120* String Storage
002130*
002140 77 MsgBoxLength
002150 01 MsgBoxText.
002160      05 MBoxText
002170      05 filler
002180 77 Fattrs-Name
002190 77 ProfileString
002200 77 ProfileName
002210
002220 77 PMFont
002230 77 ProgramINI
002240 77 ScreenINI
002250
002260 77 MainWindColor
002270
002280 77 ChildWindColor
002290
002300 77 ButtonID
002310
002320 77 PrtDataType
002330 77 Token
002340*
002350* Miscellaneous Definitions
002360*
002370 77 UT-S-SourceList
002380 77 ReturnData
002390      88 ReturnTrue
002400      88 ReturnFalse
002410 77 Loop-flag
002420      88 Program-done
002430 77 EOF
002440      88 End-Of-File
002450 77 SetFalse
002460 77 SetTrue
002470 77 ShortNull
002480 77 LongNull
002490 77 UShortNull
002500 77 ULongNull
002510 77 NullString
002520 77 ID-One
002530 77 ID-Two
002540 77 DlgReturn
002550 77 FATTR-FONTUSE-NOMIX
pic s9(9) comp-5 value 1.
pic s9(4) comp-5 value 0.
pic s9(4) comp-5 value 4.
pic s9(4) comp-5 value 0.
pic x(80).
pic x value x"00".
pic x(8) value "Courier" & x"00".
pic x(100).
pic x(11)
      value "SAMPLE.INI" & x"00".
pic x(9) value "PM_Fonts" & x"00".
pic x(9) value "COBOL-PM" & x"00".
pic x(12)
      value "ScreenTitle" & x"00".
pic x(10)
      value "MainColor" & x"00".
pic x(11)
      value "ChildColor" & x"00".
pic x(10)
      value "ID-Button" & x"00".
pic x(9) value "PM_Q_RAW" & x"00".
pic x(2) value "*" & x"00".
pic x(14) value "C:\PMCALS.TXT".
pic s9(4) comp-5.
      value 1.
      value 0.
pic x value "N".
      value "Y".
pic 9 comp-5.
      value 1.
pic 9(4) comp-5 value 0.
pic 9(4) comp-5 value 1.
pic s9(4) comp-5 value 0.
pic s9(9) comp-5 value 0.
pic 9(4) comp-5 value 0.
pic 9(9) comp-5 value 0.
pic x value x"00".
pic 9(4) comp-5 value 1.
pic 9(4) comp-5 value 2.
pic 9(4) comp-5.
pic 9(4) comp-5 value 2.

```

```
002560 77 FATTR-SEL-BOLD          pic 9(4) comp-5 value 32.
002570 77 LCID-PMC              pic s9(9) comp-5 value 0.
002580 77 LCID-Default          pic s9(9) comp-5 value 0.
002590 77 MetricsLength         pic s9(9) comp-5 value 208.
002600 77 WindowWords           pic 9(4) comp-5 value 8.
002610 77 DPDM-POSTJOBPROP     pic 9(9) comp-5 value 0.
002620*
002630* COBOL Procedure pointers
002640*
002650 77 WindowProc            procedure-pointer.
002660*
002670* Printer output storage areas
002680*
002690 01 PrinterHeading1       pic x(16) value "COBOL/PM Call = ".
002700 01 PrinterHeading2       pic x(60) value all "-".
002710 01 PrintLine.
002720      05 PrtBuffer.
002730          10 Header1        pic x(3) value x"0A0A0D".
002740          10 Header2        pic x(16).
002750          10 Header3        pic x(41).
002760      05 LineChars         pic x(2) value x"0D0A".
002770 01 PrinterDocName.
002780      05 PrtDocName        pic x(11) value "COBOL PM - ".
002790      05 PrtJobNbr         pic 99 value 0.
002800      05 filler            pic x value x"00".
002810      05 PrtDNameArray     pic x occurs 40 times.
002820 01 NbrPrtCopies.
002830      05 PrtCStatic        pic x(4) value "COP=".
002840      05 PrtCopies         pic x(3).
002850
002860*
002870* PMCall dialog data pass structure (CreateParms)
002880*
002890 01 PMCall.
002900      05 PMCLength         pic 9(4) comp-5 value 39.
002910      05 PMCType          pic 9(4) comp-5 value 0.
002920      05 PMCReturn        pic 9(4) comp-5 value 0.
002930      05 PMCLListBoxSize   pic s9(4) comp-5 value 0.
002940      05 PMCLListBoxEntry  pic x(31).
002950*
002960* Wedit dialog data pass structure (CreateParms)
002970*
002980 01 WEdit.
002990      05 WELength          pic 9(4) comp-5 value 45.
003000      05 WEType            pic 9(4) comp-5 value 0.
003010      05 WETextColor       pic s9(9) comp-5 value -1.
003020      05 WE-BN-Color       pic s9(4) comp-5 value 700.
003030      05 WEMWindowColor    pic s9(4) comp-5 value -2.
003040      05 WECWindowColor    pic s9(4) comp-5 value -2.
003050      05 WEMainTitle       pic x(31) value spaces.
003060*
```

```
003070* PMPrint dialog data pass structure (CreateParms)
003080*
003090 01 PMPrint.
003100    05 PMPLength          pic 9(4) comp-5 value 107.
003110    05 PMPType            pic 9(4) comp-5 value 0.
003120    05 PMPrtCopies        pic X(3) value spaces.
003130    05 PMPQueueName       pic x(20) value spaces.
003140    05 PMPDriverName      pic x(40) value spaces.
003150    05 PMPDrvrvName      pic x(10) value spaces.
003160    05 PMPPrtName         pic x(30) value spaces.
003170*****+
003180*      PM Call Dialog Procedures Private Storage
003190*
003200 01 PMCallPointer        usage is pointer.
003210 01 PMCallArray.
003220    05 PMCallLines         pic x(60) occurs 15 times.
003230 01 PMCallLinect        pic 9(4) comp-5 value 0.
003240*****+
003250*      Window Edit Dialog Procedures Private Storage
003260*
003270 01 WEPointer           usage is pointer.
003280*****+
003290
003300 LOCAL-STORAGE SECTION.
003310 01 Mresult             pic x(4) comp-5.
003320 01 SizeWide            pic s9(9) comp-5.
003330 01 SizeTall            pic s9(9) comp-5.
003340 01 LongWork            pic s9(9) comp-5.
003350 01 ShortWork           pic s9(4) comp-5.
003360 01 UShortWork          pic 9(4) comp-5.
003370 01 Color               pic s9(9) comp-5.
003380 01 hps                pic s9(9) comp-5.
003390 01 MaxStringLength     pic s9(4) comp-5.
003400 01 Indx                pic 9(4) comp-5.
003410 01 SelectedColor        pic s9(4) comp-5.
003420 01 KeyName              pic x(10).
003430 01 WorkString           pic x(4).
003440 01 hwndFocusWindow      pic 9(9) comp-5.
003450 01 hwndMyWindow         pic 9(9) comp-5.
003460 01 rcl.
003470    05 XLeft              pic s9(4) comp-5.
003480    05 filler              pic s9(4) comp-5.
003490    05 YBottom             pic s9(4) comp-5.
003500    05 filler              pic s9(4) comp-5.
003510    05 XRight              pic s9(4) comp-5.
003520    05 filler              pic s9(4) comp-5.
003530    05 YTop                pic s9(4) comp-5.
003540    05 filler              pic s9(4) comp-5.
003550 01 rect.
003560    05 x-coord             pic s9(9) comp-5.
003570    05 y-coord             pic s9(9) comp-5.
```

```
003580 01 SWP.
003590      05 SWP-OPTIONS          pic 9(4) comp-5.
003600      05 SWP-HEIGHT          pic 9(4) comp-5.
003610      05 SWP-WIDTH           pic 9(4) comp-5.
003620      05 SWP-Y              pic 9(4) comp-5.
003630      05 SWP-X              pic 9(4) comp-5.
003640      05 SWP-BEHIND          pic 9(9) comp-5.
003650      05 SWP-HWND           pic 9(9) comp-5.
003660*
003670* Font Attribute Table
003680*
003690 01 ATTRS.
003700      05 Fattrrs-Table.
003710      10 Fattrrs-Length        pic 9(4) comp-5.
003720      10 Fattrrs-Selection      pic 9(4) comp-5.
003730      10 Fattrrs-Match         pic 9(9) comp-5.
003740      10 Fattrrs-Facename       pic x(32).
003750      10 Fattrrs-Registry        pic 9(4) comp-5.
003760      10 Fattrrs-Codepage        pic 9(4) comp-5.
003770      10 Fattrrs-MaxBaselineExt   pic 9(9) comp-5.
003780      10 Fattrrs-AveCharWidth    pic 9(9) comp-5.
003790      10 Fattrrs-Type          pic 9(4) comp-5.
003800      10 Fattrrs-FontUse         pic 9(4) comp-5.
003810*
003820* Font Metrics Table
003830*
003840 01 FontMetrics.
003850      05 Fmetrs-Block occurs 13 times.
003860      10 Fmetrs-FamilyName      pic x(32).
003870      10 Fmetrs-FaceName        pic x(32).
003880      10 Fmetrs-Registry         pic 9(4) comp-5.
003890      10 Fmetrs-CodePage        pic 9(4) comp-5.
003900      10 Fmetrs-EmHeight         pic s9(9) comp-5.
003910      10 Fmetrs-XHeight          pic s9(9) comp-5.
003920      10 Fmetrs-MaxAscender       pic s9(9) comp-5.
003930      10 Fmetrs-MaxDescender       pic s9(9) comp-5.
003940      10 Fmetrs-LowerCaseAscent    pic s9(9) comp-5.
003950      10 Fmetrs-LowerCaseDescent   pic s9(9) comp-5.
003960      10 Fmetrs-InternalLeading    pic s9(9) comp-5.
003970      10 Fmetrs-ExternalLeading     pic s9(9) comp-5.
003980      10 Fmetrs-AveCharWidth       pic s9(9) comp-5.
003990      10 Fmetrs-MaxCharInc        pic s9(9) comp-5.
004000      10 Fmetrs-Eminc           pic s9(9) comp-5.
004010      10 Fmetrs-MaxBaseLineExt     pic s9(9) comp-5.
004020      10 Fmetrs-CharSlope          pic s9(4) comp-5.
004030      10 Fmetrs-InLineDir         pic s9(4) comp-5.
004040      10 Fmetrs-Charrot           pic s9(4) comp-5.
004050      10 Fmetrs-WeightClass        pic 9(4) comp-5.
004060      10 Fmetrs-WidthClass         pic 9(4) comp-5.
004070      10 Fmetrs-XDeviceRes        pic s9(4) comp-5.
004080      10 Fmetrs-YDeviceRes        pic s9(4) comp-5.
```

```

004090      10 Fmetrs-FirstChar           pic s9(4) comp-5.
004100      10 Fmetrs-LastChar            pic s9(4) comp-5.
004110      10 Fmetrs-DefaultChar        pic s9(4) comp-5.
004120      10 Fmetrs-BreakChar          pic s9(4) comp-5.
004130      10 Fmetrs-NominalPointSize   pic s9(4) comp-5.
004140      10 Fmetrs-MinimumPointSize   pic s9(4) comp-5.
004150      10 Fmetrs-MaximumPointSize   pic s9(4) comp-5.
004160      10 Fmetrs-Type              pic 9(4) comp-5.
004170      10 Fmetrs-Defn              pic 9(4) comp-5.
004180      10 Fmetrs-Selection         pic 9(4) comp-5.
004190      10 Fmetrs-Capabilities      pic 9(4) comp-5.
004200      10 Fmetrs-SubscriptXSize    pic s9(9) comp-5.
004210      10 Fmetrs-SubscriptYSize    pic s9(9) comp-5.
004220      10 Fmetrs-SubscriptXOffset  pic s9(9) comp-5.
004230      10 Fmetrs-SubscriptYOffset  pic s9(9) comp-5.
004240      10 Fmetrs-SuperscriptXSize  pic s9(9) comp-5.
004250      10 Fmetrs-SuperscriptYSize  pic s9(9) comp-5.
004260      10 Fmetrs-SuperscriptxOffset pic s9(9) comp-5.
004270      10 Fmetrs-SuperscriptYOffset pic s9(9) comp-5.
004280      10 Fmetrs-UnderscoreSize    pic s9(9) comp-5.
004290      10 Fmetrs-UnderscorePosition pic s9(9) comp-5.
004300      10 Fmetrs-StrikeoutSize     pic s9(9) comp-5.
004310      10 Fmetrs-StrikeoutPosition pic s9(9) comp-5.
004320      10 Fmetrs-KerningPairs     pic s9(4) comp-5.
004330      10 Fmetrs-FamilyClass      pic s9(4) comp-5.
004340      10 Fmetrs-Match           pic s9(9) comp-5.

004350*
004360* Device Open Data Structure
004370*
004380 01 DevOpenDataStruct.
004390      05 DeviceAddress           usage is pointer.
004400      05 DDriverName            usage is pointer.
004410      05 DriverData             usage is pointer.
004420      05 Redefines DriverData.
004430          10 AddrOffset          pic 9(4) comp-5.
004440          10 SelectorNbr         pic 9(4) comp-5.
004450      05 DDataType              usage is pointer.
004460      05 Comment                usage is pointer.
004470      05 DQueueName             usage is pointer.
004480      05 QueueParms            usage is pointer.
004490      05 SpoolerParms          usage is pointer.
004500      05 NetworkParms          usage is pointer.

004510
004520
004530 LINKAGE SECTION.
004540 01 hwnd                   pic 9(9) comp-5.
004550 01 Msg                    pic 9(4) comp-5.
004560 01 MsgParm1               pic 9(9) comp-5.
004570 01 Redefines MsgParm1.
004580          05 MsgParm1w1       pic 9(4) comp-5.
004590          05 MsgParm1w2       pic 9(4) comp-5.

```

```
004600 01 MsgParm2                      pic 9(9) comp-5.
004610 01 Redefines MsgParm2.
004620    05 MsgParm2w1                  pic 9(4) comp-5.
004630    05 MsgParm2w2                  pic 9(4) comp-5.
004640 01 Msg2Pointer                   usage is pointer redefines Msgparm2.
004650 01 MsgTime                      pic 9(9) comp-5.
004660 01 MsgXPoint                    pic 9(9) comp-5.
004670 01 MsgYPoint                    pic 9(9) comp-5.
004680*
004690* PMCall dialog data pass structure (CreateParms)
004700*
004710 01 DPMCall.
004720    05 DPMCLength                 pic 9(4) comp-5.
004730    05 DPMCType                  pic 9(4) comp-5.
004740    05 DPMCReturn                pic 9(4) comp-5.
004750    05 DPMCL listBoxSize          pic s9(4) comp-5.
004760    05 DPMCL listBoxEntry        pic x(31).
004770*
004780* Wedit dialog data pass structure (CreateParms)
004790*
004800 01 DWEdit.
004810    05 DWELength                 pic 9(4) comp-5.
004820    05 DWETType                  pic 9(4) comp-5.
004830    05 DWETextColor              pic s9(9) comp-5.
004840    05 DWE-BN-Color              pic s9(4) comp-5.
004850    05 DWEMWindowColor          pic s9(4) comp-5.
004860    05 DWECWindowColor          pic s9(4) comp-5.
004870    05 DWEMainTitle              pic x(31).
004880
```

```
004890 PROCEDURE DIVISION OS2API.  
004900 MAIN SECTION.  
004910* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *  
004920*  
004930* This section performs the following functions:  
004940*  
004950* Registers the program with Presentation Manager.  
004960* Establishes a message queue for this program's messages.  
004970* Registers the classes of windows to be used.  
004980* Creates the Main Window.  
004990* Performs the main message processing loop.  
005000* Destroys windows & msg queue and terminates on WM-QUIT msg.  
005010*  
005020* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *  
005030*  
005040* Initialize the Presentation Manager interface  
005050*  
005060      Call OS2API 'WinInitialize' using  
005070              by value UShortNull  
005080              returning hab  
005090*  
005100* Create a message queue for this application.  
005110*  
005120      Call OS2API 'WinCreateMsgQueue' using  
005130              by value hab  
005140              by value ShortNull  
005150              returning hmq  
005160*  
005170* Register the window classes to be used.  
005180*  
005190      Set WindowProc to ENTRY 'MainWndProc'.  
005200      Call OS2API 'WinRegisterClass' using  
005210              by value hab  
005220              by reference MainWndClass  
005230              by value WindowProc  
005240              by value CSClass  
005250              by value WindowWords  
005260              returning ReturnData  
005270  
005280      Set WindowProc to ENTRY 'ChldWndProc'.  
005290      Call OS2API 'WinRegisterClass' using  
005300              by value hab  
005310              by reference ChildWndClass  
005320              by value WindowProc  
005330              by value CSChildClass  
005340              by value UShortNull  
005350              returning ReturnData  
005360*  
005370* Load pointer to the Hand resource item.  
005380*
```

```
005390      Call OS2API 'WinLoadPointer' using
005400          by value HWND-DESKTOP
005410          by value ShortNull
005420          by value ID-Handptr
005430      returning hwndPointer
005440
005450      Call OS2API 'WinLoadPointer' using
005460          by value HWND-DESKTOP
005470          by value ShortNull
005480          by value ID-LHandptr
005490      returning hwndLPointer
005500
005510      Call OS2API 'WinLoadPointer' using
005520          by value HWND-DESKTOP
005530          by value ShortNull
005540          by value ID-DHandptr
005550      returning hwndDPointer
005560*
005570* Create the main window as a standard window.
005580*
005590      Call OS2API 'WinCreateStdWindow' using
005600          by value HWND-DESKTOP
005610          by value ULongNull
005620          by reference FCF-MAIN
005630          by reference MainWndClass
005640          by reference NullString
005650          by value ULongNull
005660          by value UShortNull
005670          by value ID-MainWind
005680          by reference hwndClient
005690      returning hwndFrame
005700*
005710* Subclass the Main Window Frame
005720*
005730      Set WindowProc to ENTRY 'FrameProc'
005740      Call OS2API 'WinSubclassWindow' using
005750          by value hwndFrame
005760          by value WindowProc
005770      returning WindowProc
005780*
005790* Save the addr of the default procedure & Client window handle
005800*
005810      Call OS2API 'WinSetWindowULong' using
005820          by value hwndFrame
005830          by value QWL-OldProc
005840          by value WindowProc
005850      returning ReturnData
005860
```

```
005870     Call OS2API 'WinSetWindowULong' using
005880             by value    hwndFrame
005890             by value    QWL-hwndWindow
005900             by value    hwndClient
005910             returning  ReturnData
005920
005930*
005940* Test for the Courier bit-mapped font
005950*
005960     Move Fattrs-Name to KeyName
005970     Call OS2API 'WinUpper' using
005980             by value    hab
005990             by value    ShortNull
006000             by value    ShortNull
006010             by reference Keyname
006020             returning   ReturnData
006030
006040     Call OS2API 'PrfQueryProfileString' using
006050             by value    HINI-USERPROFILE
006060             by reference PMFont
006070             by reference KeyName
006080             by value    LongNull
006090             by reference ProfileString
006100             by value    66 size 4
006110             returning   ReturnData
006120
006130     If ReturnData = 0
006140*
006150* If RC = 0, Courier bit-mapped font not installed
006160*
006170     Move 80 to MaxStringLength
006180     Call OS2API 'WinLoadString' using
006190             by value    hab
006200             by value    ShortNull
006210             by value    IDS-MsgBoxTwo
006220             by value    MaxStringLength
006230             by reference MsgBoxText
006240             returning   MsgBoxLength
006250
006260     Call OS2API 'WinMessageBox' using
006270             by value    HWND-DESKTOP
006280             by value    hwndClient
006290             by reference MsgBoxText
006300             by reference NullString
006310             by value    ID-Two
006320             by value    MB-CRITICAL
006330             returning   Mresult
006340
006350     Set program-done to true
006360
006370     Else
```

```
006380*
006390* Open the program's profile
006400*
006410      Call OS2API 'PrfOpenProfile' using
006420              by value      hab
006430              by reference ProfileName
006440              returning    Hini-MyProfile
006450*
006460* Query OS2.INI for Main window title
006470*
006480      Call OS2API 'PrfQueryProfileString' using
006490              by value      Hini-MyProfile
006500              by reference ProgramINI
006510              by reference ScreenINI
006520              by value      LongNull
006530              by reference WEMainTitle
006540              by value      31  size 4
006550              returning   ReturnData
006560
006570      If ReturnData > 0
006580*
006590* If entry found, make it the Main window title
006600*
006610      Call OS2API 'WinSetWindowText' using
006620              by value      hwndFrame
006630              by reference WEMainTitle
006640              returning   ReturnData
006650      Move spaces to WEMainTitle
006660      End-if
006670*
006680* Query OS2.INI for Main window color
006690*
006700      Call OS2API 'PrfQueryProfileInt' using
006710              by value      Hini-MyProfile
006720              by reference ProgramINI
006730              by reference MainWindColor
006740              by value      ShortNull
006750              returning   ReturnData
006760
006770      If ReturnData > 0
006780*
006790* Account for negative colors
006800*
006810      If ReturnData > 15
006820          Compute ReturnData = ReturnData - 18
006830      End-if
006840      Move ReturnData to WEMwindowColor
006850      End-if
006860*
006870* Query OS2.INI for Child window color
006880*
```

```
006890      Call OS2API 'PrfQueryProfileInt' using
006900          by value      Hini-MyProfile
006910          by reference ProgramINI
006920          by reference ChildWindColor
006930          by value      ShortNull
006940          returning    ReturnData
006950
006960      If ReturnData > 0
006970*
006980* Account for negative colors
006990*
007000          If ReturnData > 15
007010              Compute ReturnData = ReturnData - 18
007020          End-if
007030          Move ReturnData to WECWindowColor
007040          End-if
007050*
007060* Determine child window text color
007070*
007080          Move Clr-Black to WETTextColor
007090          If (WECWindowColor > Clr-Yellow and
007100              WECWindowColor < Clr-PaleGray) or
007110              WECWindowColor = Clr-Blue
007120          Move Clr-White to WETTextColor
007130          End-if
007140*
007150* Query OS2.INI for the last color radio button selected
007160*
007170          Call OS2API 'PrfQueryProfileInt' using
007180              by value      Hini-MyProfile
007190              by reference ProgramINI
007200              by reference ButtonID
007210              by value      ShortNull
007220              returning    ReturnData
007230
007240          If ReturnData > 0
007250              Move ReturnData to WE-BN-Color
007260          End-if
007270*
007280* Measure the size of the dispaly screen
007290*
007300          Call OS2API 'WinQuerySysValue' using
007310              by value      HWND-DESKTOP
007320              by value      SV-CXSCREEN
007330              returning    SizeWide
007340
007350          Call OS2API 'WinQuerySysValue' using
007360              by value      HWND-DESKTOP
007370              by value      SV-CYSCREEN
007380              returning    SizeTall
007390*
```

```
007400* Size the Main Window to 70% high by 80% wide as the desktop
007410* and center on the Desktop
007420*
007430      compute YTop      = SizeTall * .7
007440      compute XRight   = SizeWide * .8
007450      compute XLeft    = (SizeWide - XRight) / 2
007460      compute YBottom  = (SizeTall - YTop) / 2
007470*
007480* Display the Main Window
007490*
007500      Call OS2API 'WinSetWindowPos' using
007510          by value hwndFrame
007520          by value HWND-TOP
007530          by value XLeft
007540          by value YBottom
007550          by value XRight
007560          by value YTop
007570          by value SWP-WINDOW
007580          returning ReturnData
007590*
007600* Get the handle of the Main Window menu
007610*
007620      Call OS2API 'WinWindowFromID' using
007630          by value hwndFrame
007640          by value FID-MENU
007650          returning hwndMenu
007660*
007670* Get the next message for this application or wait until a
007680* message is posted to the Application Message Queue.
007690*
007700      If hwndFrame not = 0
007710          perform until program-done
007720
007730      Call OS2API 'WinGetMsg' using
007740          by value hab
007750          by reference QMSG
007760          by value ULongNull
007770          by value UShortNull
007780          by value UShortNull
007790          returning ReturnData
007800*
007810* If WM-QUIT message received perform the closedown routine.
007820*
007830      If (QMSG-MSGID = WM-QUIT)
007840          Set WindowProc to ENTRY 'TermDlgProc'
```

```
007850      Call OS2API 'WinDlgBox' using
007860          by value HWND-DESKTOP
007870          by value hwndFrame
007880          by value WindowProc
007890          by value ShortNull
007900          by value ID-TermDlgBox
007910          by value LongNull
007920          returning DlgReturn
007930
007940      If DlgReturn = ID-TermOK
007950          Set program-done to true
007960      end-if
007970      else
007980*
007990* Send all messages to window default processing routine
008000*
008010      Call OS2API 'WinDispatchMsg' using
008020          by value hab
008030          by reference QMSG
008040          returning ReturnData
008050
008060      end-if
008070      end-perform
008080  End-If.
008090*
008100* Close the user profile
008110*
008120      Call OS2API 'PrfCloseProfile' using
008130          by value Hini-MyProfile
008140          returning ReturnData
008150*
008160* Destroy all remaining windows - (Main and all children)
008170*
008180      Call OS2API 'WinDestroyWindow' using
008190          by value hwndFrame
008200          returning ReturnData
008210*
008220* Destroy the Application Message Queue.
008230*
008240      Call OS2API 'WinDestroyMsgQueue' using
008250          by value hmq
008260          returning ReturnData
008270*
008280* Terminate this program use of Presentation Manager services.
008290*
008300      Call OS2API 'WinTerminate' using
008310          by value hab
008320          returning ReturnData
008330  STOP RUN.
008340*****008350
```

```
008360 FrameProc section.  
008370* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *  
008380* Frame Window Procedure: FrameProc. This procedure *  
008390* intercepts the MouseMove messages for the Main Frame window *  
008400* and determines which pointer to display based upon the *  
008410* frame location being touched. *  
008420* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *  
008430 entry 'FrameProc' using by value hwnd  
008440                      by value Msg  
008450                      by value MsgParm1  
008460                      by value MsgParm2  
008470*  
008480* Set return code to OK and evaluate the message  
008490*  
008500      move 0 to Mresult  
008510      evaluate Msg  
008520  
008530          when WM-MOUSEMOVE  
008540              Call OS2API 'WinQueryFocus' using  
008550                  by value HWND-DESKTOP  
008560                  by value ShortNull  
008570                  returning hwndFocusWindow  
008580  
008590          Call OS2API 'WinQueryWindowULong' using  
008600                  by value hwnd  
008610                  by value QWL-hwndWindow  
008620                  returning hwndMyWindow  
008630  
008640          If hwndFocusWindow = hwndMyWindow  
008650*  
008660* Determine the size of the frame window  
008670*  
008680          Call OS2API 'WinQueryWindowRect' using  
008690                  by value hwnd  
008700                  by reference rcl  
008710                  returning ReturnData  
008720*  
008730* Determine the width of the border and set border locations  
008740*  
008750          Call OS2API 'WinQuerySysValue' using  
008760                  by value HWND-DESKTOP  
008770                  by value SV-CXSIZEBORDER  
008780                  returning ReturnData  
008790  
008800          Subtract ReturnData from YTop giving YTop  
008810          If MsgParm1w2 is not less than YTOP or  
008820              MsgParm1w2 is not greater than ReturnData  
008830              Move hwndDPointer to LongWork  
008840          Else  
008850              Move hwndLPointer to LongWork  
008860          End-if
```

```
008870
008880      Call OS2API 'WinSetPointer' using
008890          by value HWND-DESKTOP
008900          by value LongWork
008910          returning ReturnData
008920      End-if
008930
008940      when other
008950          Call OS2API 'WinQueryWindowULong' using
008960              by value hwndFrame
008970              by value QWL-OldProc
008980              returning WindowProc
008990
009000          Call OS2API WindowProc using
009010              by value hwnd
009020              by value Msg
009030              by value MsgParm1
009040              by value MsgParm2
009050              returning Mresult
009060      End-evaluate.
009070      exit program returning Mresult.
009080*****
009090
```

```
009100 MainWndProc section.  
009110* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *  
009120* Main Window Procedure: MainWndProc. *  
009130* Receives and processes all messages sent to the program's *  
009140* Main Window's client window *  
009150* Messages are passed on entry in the standard message format.*  
009160* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *  
009170* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *  
009180 entry 'MainWndProc' using by value hwnd  
009190           by value Msg  
009200           by value MsgParm1  
009210           by value MsgParm2  
009220*  
009230* Set return code to OK and evaluate the message  
009240*  
009250     move 0 to Mresult  
009260     evaluate Msg  
009270  
009280           when WM-MOUSEMOVE  
009290*  
009300* Check for WM-MOUSEMOVE message  
009310*  
009320     Call OS2API 'WinQueryFocus' using  
009330           by value HWND-DESKTOP  
009340           by value ShortNull  
009350           returning hwndFocusWindow  
009360  
009370     If hwndFocusWindow = hwnd  
009380         Call OS2API 'WinSetPointer' using  
009390             by value HWND-DESKTOP  
009400             by value hwndPointer  
009410             returning ReturnData  
009420     Else  
009430         Call OS2API 'WinDefWindowProc' using  
009440             by value hwnd  
009450             by value Msg  
009460             by value MsgParm1  
009470             by value MsgParm2  
009480             returning Mresult  
009490     End-if  
009500  
009510     when WM-PAINT  
009520*  
009530* Check for WM-PAINT message  
009540*  
009550     Move 0 to XLeft YBottom XRight Ytop  
009560     Call OS2API 'WinBeginPaint' using  
009570           by value hwnd  
009580           by value LongNull  
009590           by reference rcl  
009600           returning hps
```

```

009610
009620      Move WEMWindowColor to Color
009630      Call OS2API 'WinFillRect' using
009640          by value     hps
009650          by reference rcl
009660          by value     Color
009670          returning   ReturnData
009680
009690          Call OS2API 'WinEndPaint' using
009700              by value   hps
009710              returning ReturnData
009720*
009730* Check for WM-COMMAND messages
009740*
009750      when WM-COMMAND
009760*
009770* Check for the MI-Exit menu item
009780*
009790      If MsgParm1w1 = MI-Exit
009800*
009810* Post the WM-Quit message to this program's message queue
009820*
009830      Call OS2API 'WinPostMsg' using
009840          by value     hwndFrame
009850          by value     WM-QUIT
009860          by value     LongNull
009870          by value     LongNull
009880          returning   ReturnData
009890
009900      End-If
009910*
009920* Check for MI-Print menu item
009930*
009940      If MsgParm1w1 = MI-Print
009950
009960      Move hwndClient to MsgParm1
009970      Set Msg2Pointer to address of PMPrint
009980      Call OS2API 'PrtSampl' using
009990          by value     MsgParm1
010000          by value     MsgParm2
010010          returning   DlgReturn
010020
010030      If DlgReturn = ID-PrinterButtonCan
010040          exit program returning Mresult
010050      End-if
010060*
010070* Retrieve size of DevOpenData structure
010080*

```

```
010090          Call OS2API 'DevPostDeviceModes' using
010100                  by value      hab
010110                  by value      LongNull
010120                  by reference PMPDrvrName
010130                  by reference PMPPrtName
010140                  by value      LongNull
010150                  by value      DPDM-POSTJOBPROP
010160                  returning    UShortWork
010170*
010180* Allocate memory for the driver data structure
010190*
010200          Move 0 to AddrOffset
010210          Call OS2API 'DosAllocSeg' using
010220                  by value      UShortWork
010230                  by reference SelectorNbr
010240                  by value      0 size 2
010250                  returning    ReturnData
010260*
010270* Display the selected printer driver job properties
010280*
010290          Call OS2API 'DevPostDeviceModes' using
010300                  by value      hab
010310                  by value      DriverData
010320                  by reference PMPDrvrName
010330                  by reference PMPPrtName
010340                  by value      LongNull
010350                  by value      DPDM-POSTJOBPROP
010360                  returning    ReturnData
010370*
010380* Set field addresses in DevOpenDataStructure
010390*
010400          Set DeviceAddress to address of PMPQueueName
010410          Set DDriverName to address of PMPDriverName
010420          Set DDataType to address of PrtDataType
010430          Set Comment to address of NullString
010440          Set DQueueName to address of NullString
010450          Set QueueParms to address of NbrPrtCopies
010460          Move PMPrtCopies to PrtCopies
010470          Call OS2API 'SplQmOpen' using
010480                  by reference Token
010490                  by value      7 size 4
010500                  by reference DevOpenDataStruct
010510                  returning    hspl
010520
010530          add 1 to PrtJobNbr giving PrtJobNbr
010540          Call OS2API 'SplQmStartDoc' using
010550                  by value      hspl
010560                  by reference PrinterDocName
010570                  returning    ReturnData
010580
```

```
010590          Perform 100-Print-Heading
010600          Move 62 to LongWork
010610          Perform 110-Print-Body with test before
010620              varying Indx from 1 by 1
010630              until Indx > PMCallLineCt
010640*
010650* End the output, close the spool and free the acquired memory
010660*
010670          Call OS2API 'SplQmEndDoc' using
010680              by value hspl
010690              returning ReturnData
010700
010710          Call OS2API 'SplQmClose' using
010720              by value hspl
010730              returning ReturnData
010740
010750          Call OS2API 'DosFreeSeg' using
010760              by value SelectorNbr
010770              returning ReturnData
010780
010790          End-if
010800*
010810* Check for the MI-Source menu item
010820*
010830          If MsgParm1w1 = MI-Source
010840              Set WindowProc to ENTRY 'PMcallDlgProc'
010850              Call OS2API 'WinDlgBox' using
010860                  by value      HWND-DESKTOP
010870                  by value      hwndClient
010880                  by value      WindowProc
010890                  by value      ShortNull
010900                  by value      ID-PMCallDlgBox
010910                  by reference PMCall
010920                  returning    PMCRReturn
010930
010940          If PMCRReturn = ID-PMCallButtonCan
010950              exit program returning Mresult
010960          End-if
010970
010980*
010990* Create the child window as a standard window.
011000*
```

```
011010          Call OS2API 'WinCreateStdWindow' using
011020              by value      hwnd
011030              by value      ULongNull
011040              by reference FCF-CHILD
011050              by reference ChildWndClass
011060              by reference NullString
011070              by value      ULongNull
011080              by value      USHORTNULL
011090              by value      ID-SourceWind
011100              by reference hwndSourceClient
011110              returning    hwndSource
011120*
011130* Set the selected call name as the child window title
011140*
011150          Call OS2API 'WinSetWindowText' using
011160              by value      hwndSource
011170              by reference PMCLListBoxEntry
011180              returning    ReturnData
011190*
011200* Remove null terminating character for compare, load PM call
011210*
011220          Inspect PMCLListBoxEntry replacing first x"00"
011230              by space
011240          Perform 150-Locate-Call
011250*
011260* Calculate and set size and position of child window
011270*
011280          Call OS2API 'WinQueryWindowRect' using
011290              by value      hwnd
011300              by reference rcl
011310              returning    ReturnData
011320
011330          compute XLeft = XLeft + 10
011340          compute YBottom = YBottom + 10
011350          compute XRight = XRight - 20
011360          compute YTop = YTop - 20
011370          Call OS2API 'WinSetWindowPos' using
011380              by value      hwndSource
011390              by value      HWND-TOP
011400              by value      XLeft
011410              by value      YBottom
011420              by value      XRight
011430              by value      YTop
011440              by value      SWP-WINDOW
011450              returning    ReturnData
011460*
011470* Disable Source entry from Menu until Source Window ends
011480*
011490          Move MI-Source to MsgParm1w1
011500          Move SetTrue to MsgParm1w2
011510          Move MIA-DISABLED to MsgParm2w1 MsgParm2w2
```

```

011520          Call OS2API 'WinSendMsg' using
011530                  by value hwndMenu
011540                  by value MM-SETITEMATTR
011550                  by value MsgParm1
011560                  by value MsgParm2
011570                  returning ReturnData
011580*
011590* Enable the Print menu entry while Source Window active
011600*
011610          Move MI-Print to MsgParm1w1
011620          Move SetTrue to MsgParm1w2
011630          Move MIA-DISABLED to MsgParm2w1
011640          Move 0 to MsgParm2w2
011650          Call OS2API 'WinSendMsg' using
011660                  by value hwndMenu
011670                  by value MM-SETITEMATTR
011680                  by value MsgParm1
011690                  by value MsgParm2
011700                  returning ReturnData
011710
011720          End-If
011730*
011740* Check for MI-Wedit menu item
011750*
011760          If MsgParm1w1 = MI-Wedit
011770                  Set WindowProc to ENTRY 'WEditDlgProc'
011780                  Call OS2API 'WinDlgBox' using
011790                      by value      HWND-DESKTOP
011800                      by value      hwndClient
011810                      by value      WindowProc
011820                      by value      ShortNull
011830                      by value      ID-WinEditDlgBox
011840                      by reference WEdit
011850                      returning     DlgReturn
011860
011870          If DlgReturn = ID-WinDlgItemCan
011880                  Exit program returning Mresult
011890          End-if
011900*
011910* Set new main window title if text returned
011920*
011930          If WEMainTitle is not equal to spaces
011940                  Call OS2API 'WinSetWindowText' using
011950                      by value      hwndFrame
011960                      by reference WEMainTitle
011970                      returning     ReturnData
011980*
011990* Write new Main Window title to the program's profile
012000*

```

```
012010          Call OS2API 'PrfWriteProfileString' using
012020                  by value      Hini-MyProfile
012030                  by reference ProgramINI
012040                  by reference ScreenINI
012050                  by reference WEMainTitle
012060                  returning    ReturnData
012070
012080          Move spaces to WEMainTitle
012090          End-if
012100*
012110* If any color changed, save last button checked
012120*
012130          If WEMWindowColor is not equal to 0 or
012140          WECWindowColor is not equal to 0
012150          Move WE-BN-Color to WorkString
012160          Call OS2API 'PrfWriteProfileData' using
012170                  by value      Hini-MyProfile
012180                  by reference ProgramINI
012190                  by reference ButtonID
012200                  by reference WorkString
012210                  by value      4 size 4
012220                  returning    ReturnData
012230          End-if
012240*
012250* Set new main or child window color if window checked
012260*
012270          If WEMWindowColor is not equal to 0
012280
012290          Call OS2API 'WinInvalidateRegion' using
012300                  by value      hwnd
012310                  by value      LongNull
012320                  by value      SetFalse
012330                  returning    ReturnData
012340*
012350* Save new Main Window color in the program's profile
012360*
012370          If WEMWindowColor = -2
012380          Move 16 to WorkString
012390          Else
012400          Move WEMWindowColor to WorkString
012410          End-if
012420          Call OS2API 'PrfWriteProfileData' using
012430                  by value      Hini-MyProfile
012440                  by reference ProgramINI
012450                  by reference MainWindColor
012460                  by reference WorkString
012470                  by value      4 size 4
012480                  returning    ReturnData
012490          End-if
012500          If WECWindowColor is not equal to 0
012510*
```

```
012520* Save new child Window color in the program's profile
012530*
012540          If WECWindowColor = -2
012550              Move 16 to WorkString
012560          Else
012570              Move WECWindowColor to WorkString
012580          End-if
012590      Call OS2API 'PrfWriteProfileData' using
012600                  by value Hini-MyProfile
012610                  by reference ProgramINI
012620                  by reference ChildWindColor
012630                  by reference WorkString
012640                  by value        4 size 4
012650                  returning     ReturnData
012660
012670      Call OS2API 'WinBeginEnumWindows' using
012680                  by value hwndClient
012690                  returning hwndChildWindows
012700
012710          Perform with test after
012720              until hwndChildQuery is equal to 0
012730*
012740* Begin enumeration of immediate child windows of hwndClient
012750*
012760      Call OS2API 'WinGetNextWindow' using
012770                  by value hwndChildWindows
012780                  returning hwndChildQuery
012790
012800      If hwndChildQuery is not equal to 0
012810          Call OS2API 'WinBeginEnumWindows' using
012820                  by value hwndChildQuery
012830                  returning hwndsWindow
012840
012850          Perform with test after
012860              until hwndsWindowQuery = 0
012870*
012880* If children found enumerate all of their children
012890*
012900      Call OS2API 'WinGetNextWindow' using
012910                  by value hwndsWindow
012920                  returning hwndsWindowQuery
012930
012940      If hwndsWindowQuery is not equal to 0
012950          Call OS2API 'WinInvalidateRegion' using
012960                  by value hwndsWindowQuery
012970                  by value LongNull
012980                  by value SetFalse
012990                  returning ReturnData
013000
013010      End-if
End-Perform
```

```
013020          Call OS2API 'WinEndEnumWindows' using
013030                      by value hwndSWindow
013040                      returning ReturnData
013050      End-if
013060      End-perform
013070          Call OS2API 'WinEndEnumWindows' using
013080                      by value hwndChildWindows
013090                      returning ReturnData
013100      End-if
013110      End-if
013120*
013130* Check for the MI-About menu item
013140*
013150      If MsgParm1w1 = MI-About
013160          Move 80 to MaxStringLength
013170          Call OS2API 'WinLoadString' using
013180                      by value hab
013190                      by value ShortNull
013200                      by value IDS-MsgBoxOne
013210                      by value MaxStringLength
013220                      by reference MsgBoxText
013230                      returning MsgBoxLength
013240
013250          Call OS2API 'WinMessageBox' using
013260                      by value HWND-DESKTOP
013270                      by value hwndClient
013280                      by reference MsgBoxText
013290                      by reference NullString
013300                      by value ID-One
013310                      by value MB-INFO-OK
013320                      returning Mresult
013330      End-If
013340*
013350* Return all other messages for default processing
013360*
013370      when other
013380
013390          Call OS2API 'WinDefWindowProc' using
013400                      by value hwnd
013410                      by value Msg
013420                      by value MsgParm1
013430                      by value MsgParm2
013440                      returning Mresult
013450
013460      End-evaluate
013470      exit program returning Mresult.
013480*****
013490* 100-Print-Heading. This procedure prints the heading lines
013500*      for the PM Call printer output
013510*****
```

```
013520 100-Print-Heading.  
013530      Move PrinterHeading1 to Header2  
013540      Move PMCLListBoxEntry to Header3  
013550      Move 62 to LongWork  
013560      Perform 120-PrtWrite  
013570  
013580      Move PrinterHeading2 to PrtBuffer  
013590      Perform 120-PrtWrite.  
013600  
013610      Move LineChars to Header1  
013620      Move 2 to LongWork  
013630      Perform 120-PrtWrite.  
013640*****  
013650* 110-Print-Body. This procedure prints the selected OS2 call  
013660*      as found in the PMCallArray table.  
013670*****  
013680 110-Print-Body.  
013690      Move PMCallLines(Indx) to PrtBuffer  
013700      Perform 120-PrtWrite.  
013710*****  
013720* 120-PrtWrite. This procedure sends the actual printer output  
013730*      lines to the Print Manager spooler  
013740*****  
013750 120-PrtWrite.  
013760      Call OS2API 'SplQmWrite' using  
013770          by value      hspl  
013780          by value      LongWork  
013790          by reference Printline  
013800          returning     ReturnData.  
013810*****  
013820* 150-Locate-Call. This procedure searches the PMCALL.TXT file  
013830*      for the source code of the PM call selected by the user  
013840*      during the PMCall dialog.  
013850*****  
013860 150-Locate-Call.  
013870      Open input Source-Listing  
013880      Move 0 to EOF PMCallLineCt  
013890      Perform 160-Record-Read until End-Of-File  
013900      Close Source-Listing.  
013910  
013920 160-Record-Read.  
013930      Read Source-Listing into SourceRecord  
013940          At end move SetTrue to EOF.  
013950      If REC-ID is equal to PMCLListBoxEntry  
013960          Perform 170-Source-Load until End-Of-File  
013970      End-if.  
013980  
013990 170-Source-Load.  
014000      Read Source-Listing  
014010      If Statement is equal to " "  
014020          Move SetTrue to EOF
```

```
014030      Else
014040          Add 1 to PMCallLineCt giving PMCallLineCt
014050          Move Statement to PMCallLines(PMCallLineCt)
014060      End-if.
014070*****
014080
```

```
014090 ChldWndProc section.
014100* * * * * * * * * * * * * * * * * * * * * * *
014110* Child Window Procedure: ChldWndProc.
014120*   Receives and processes all messages sent to windows with *
014130*   a class of "ChldWndProc".
014140*   Messages are passed on entry in the standard message format.* *
014150*
014160* * * * * * * * * * * * * * * * * * * * * * *
014170 entry 'ChldWndProc' using by value hwnd
014180           by value Msg
014190           by value MsgParm1
014200           by value MsgParm2
014210*
014220* Set return code to OK and evaluate the message
014230*
014240     move 0 to Mresult
014250     evaluate Msg
014260*
014270* Check for menu messages
014280*
014290     when WM-MOUSEMOVE
014300       Call OS2API 'WinSetPointer' using
014310           by value HWND-DESKTOP
014320           by value hwndPointer
014330           returning ReturnData
014340
014350     when WM-COMMAND
014360       If MsgParm1w1 = MI-Closesw
014370         Call OS2API 'WinDestroyWindow' using
014380             by value hwndSource
014390             returning ReturnData
014400*
014410* Enable the Source menu entry
014420*
014430     Move MI-Source to MsgParm1w1
014440     Move SetTrue to MsgParm1w2
014450     Move MIA-DISABLED to MsgParm2w1
014460     Move 0 to MsgParm2w2
014470     Call OS2API 'WinSendMsg' using
014480         by value hwndMenu
014490         by value MM-SETITEMATTR
014500         by value MsgParm1
014510         by value MsgParm2
014520         returning ReturnData
014530     Move 0 to hwndSource hwndSourceClient
014540*
014550* Disable the Print menu entry
014560*
014570     Move MI-PRINT to MsgParm1w1
014580     Move SetTrue to MsgParm1w2
```

```
014590      Move MIA-DISABLED to MsgParm2w1 MsgParm2w2
014600      Call OS2API 'WinSendMsg' using
014610          by value hwndMenu
014620          by value MM-SETITEMATTR
014630          by value MsgParm1
014640          by value MsgParm2
014650          returning ReturnData
014660      end-if
014670*
014680* Check for WM-PAINT message
014690*
014700      when WM-PAINT
014710          Move 0 to XLeft YBottom XRight Ytop
014720          Call OS2API 'WinBeginPaint' using
014730              by value hwnd
014740              by value LongNull
014750              by reference rcl
014760              returning hps
014770
014780          Move WECWindowColor to Color
014790          Call OS2API 'WinFillRect' using
014800              by value hps
014810              by reference rcl
014820              by value Color
014830
014840          If (PMCallLineCt > 0)
014850              Perform 300-Screen-Write
014860          End-if
014870
014880          Call OS2API 'WinEndPaint' using
014890              by value hps
014900*
014910* Return all other messages for default processing
014920*
014930      when other
014940          Call OS2API 'WinDefWindowProc' using
014950              by value hwnd
014960              by value Msg
014970              by value MsgParm1
014980              by value MsgParm2
014990              returning Mresult
015000      End-evaluate
015010      exit program returning Mresult.
015020*****
015030* 300-Screen-Write. This perform builds the Courier logical
015040* font, set the correct font, calls 310-Screen-Out to write
015050* the characters to the Presentation Space then resets the
015060* current font to the default font
015070*****
015080 300-Screen-Write.
015090*
```

```
015100* Determine density of display screen
015110*
015120    Call OS2API 'WinQuerySysValue' using
015130        by value HWND-DESKTOP
015140        by value SV-CXSCREEN
015150        returning SizeWide
015160*
015170* Query for all Courier image fonts
015180*
015190    Move 13 to LongWork
015200    Call OS2API 'GpiQueryFonts' using
015210        by value hps
015220        by value QF-PUBLIC
015230        by reference Fattrs-Name
015240        by reference LongWork
015250        by value MetricsLength
015260        by reference FontMetrics
015270        returning ReturnData
015280
015290    Initialize ATTRS replacing numeric by 0
015300    Perform 310-Font-Query
015310        varying indx from 1 by 1 until indx > LongWork
015320
015330    If Fattrs-Match > 0
015340        Move 56 to Fattrs-Length
015350        Move FATTR-FONTUSE-NOMIX to Fattrs-Fontuse
015360        Move 1 to LCID-PMC
015370        Call OS2API 'GpiCreateLogFont' using
015380            by value hps
015390            by reference Fattrs-Name
015400            by value LCID-PMC
015410            by reference Attrs
015420            returning ReturnData
015430
015440    Call OS2API 'GpiSetCharSet' using
015450        by value hps
015460        by value LCID-PMC
015470        returning ReturnData
015480    End-if
015490
015500    Call OS2API 'GpiQueryFontMetrics' using
015510        by value hps
015520        by value MetricsLength
015530        by reference FontMetrics
015540        returning ReturnData
015550
015560    Call OS2API 'GpiSetColor' using
015570        by value hps
015580        by value WETextColor
015590        returning ReturnData
015600
```

```
015610      Call OS2API 'WinQueryWindowRect' using
015620          by value      hwnd
015630          by reference rcl
015640          returning    ReturnData
015650
015660      Move 10 to x-coord of rect
015670      Move 60 to LongWork
015680      Perform 320-Screen-Out
015690          varying indx from 1 by 1 until indx > PMCallLineCt
015700
015710      Call OS2API 'GpiSetCharSet' using
015720          by value      hps
015730          by value      LCID-Default
015740          returning    ReturnData
015750
015760      Call OS2API 'GpiDeleteSetId' using
015770          by value      hps
015780          by value      LCID-PMC
015790          returning    ReturnData.
015800*****
015810* 310-Font-Query. This perform searches for the best Courier
015820*     image from all the Courier fonts returned by PM.
015830*****
015840 310-Font-Query.
015850      If SizeWide > 640
015860*
015870* If 8514/A or XGA adapter select 12 point Courier Bold
015880*
015890      If Fmetrs-NominalPointSize(Indx) = 120
015900          Move FATTR-SEL-BOLD to Fattrss-Selection
015910          If Fattrss-Match = 0
015920              Move Fmetrs-Match(Indx) to Fattrss-Match
015930              Move Fmetrs-FaceName(Indx) to Fattrss-FaceName
015940          End-if
015950          If (Fmetrs-MaxBaseLineExt(Indx) = 15) and
015960              (Fmetrs-AveCharWidth(Indx) = 12)
015970              Move Fmetrs-Match(Indx) to Fattrss-Match
015980              Compute Indx = LongWork + 1
015990          End-if
016000      End-if
016010      Else
016020*
016030* If VGA or lower select 10 point Courier
016040*
016050      If Fmetrs-NominalPointSize(Indx) = 100
016060          If Fattrss-Match = 0
016070              Move Fmetrs-Match(Indx) to Fattrss-Match
016080              Move Fmetrs-FaceName(Indx) to Fattrss-FaceName
016090          End-if
016100          If (Fmetrs-MaxBaseLineExt(Indx) = 12) and
016110              (Fmetrs-AveCharWidth(Indx) = 9)
```

```
016120      Move Fmetrs-Match(Indx) to Fattrs-Match
016130      Compute Indx = LongWork + 1
016140      End-if
016150      End-if
016160      End-if.
016170*****
016180* 320-Screen-Out. This perform determines the vertical location
016190*      for each output line and places the characters in the
016200*      Presentation Space.
016210*****
016220 320-Screen-Out.
016230      Compute y-coord = (YTOP - 10) -
016240              Indx * (Fmetrs-MaxBaseLineExt(1) +
016250              Fmetrs-ExternalLeading(1))
016260
016270      Call OS2API 'GpiCharStringAt' using
016280          by value      hps
016290          by reference rect
016300          by value      LongWork
016310          by reference PMCallLines(Indx)
016320          returning    ReturnData.
016330*****
016340
```

```
016350 TermDlgProc section.  
016360*****  
016370* Terminate Dialog Procedure: TermDlgProc.  
016380*   Processes the Terminate Dialog Box displayed when the user  
016390*   selects Exit or Close to end the sample application.  
016400*****  
016410 entry 'TermDlgProc' using by value hwnd  
016420           by value msg  
016430           by value MsgParm1  
016440           by value MsgParm2.  
016450*  
016460* Set return code to OK and evaluate the message  
016470*  
016480     move 0 to Mresult  
016490     evaluate Msg  
016500*  
016510* Process dialog messages  
016520*  
016530     when WM-COMMAND  
016540       Move MsgParm1w1 to DlgReturn  
016550       Call OS2API 'WinDismissDlg' using  
016560           by value hwnd  
016570           by value DlgReturn  
016580*  
016590* Return all other messages for default processing  
016600*  
016610     when other  
016620       Call OS2API 'WinDefDlgProc' using  
016630           by value hwnd  
016640           by value msg  
016650           by value MsgParm1  
016660           by value MsgParm2  
016670           returning Mresult  
016680     End-evaluate  
016690     exit program returning Mresult.  
016700*****  
016710
```

```

016720 PMCallDlgProc section.
016730 ****
016740* PMCall Dialog Procedure: PMCallDlgProc.
016750* Determines which of the 42 PM API calls the user wishes to
016760* display. This routine loads the PMCall list box and returns
016770* the selected ID to the program.
016780 ****
016790 entry 'PMCallDlgProc' using by value hwnd
016800                                by value msg
016810                                by value MsgParm1
016820                                by value MsgParm2.
016830*
016840* Set return code to OK and evaluate the message
016850*
016860      move 0 to Mresult
016870      evaluate Msg
016880*
016890* Process dialog messages
016900*
016910      when WM_INITDLG
016920*
016930* Save the pointer to the PMCall data pass area
016940*
016950      Set PMCallPointer to Msg2Pointer
016960      Set Address of DPMCall to PMCallPointer
016970*
016980* Erase any entries in the PM Call list box
016990*
017000      Call OS2API 'WinSendDlgItemMsg' using
017010          by value hwnd
017020          by value ID-PMCallListBox
017030          by value LM-DELETEALL
017040          by value LongNull
017050          by value LongNull
017060          returning ReturnData
017070*
017080* Fill the list box with call titles
017090*
017100      Compute UShortWork = IDS-PMCall1 - 1
017110      Perform 400-Fill-Box
017120          with test after
017130          until DPMCL listBoxSize = 0
017140*
017150* Preselect the first item and center the dialog box
017160*

```

```
017170      Call OS2API 'WinSendDlgItemMsg' using
017180          by value hwnd
017190          by value ID-PMCallListBox
017200          by value LM-SELECTITEM
017210          by value 0 size 4
017220          by value 1 size 4
017230          returning ReturnData
017240
017250      Call OS2API 'WinQueryWindowRect' using
017260          by value HWND-DESKTOP
017270          by reference rcl
017280          returning ReturnData
017290
017300      Call OS2API 'WinQueryWindowPos' using
017310          by value hwnd
017320          by reference SWP
017330          returning ReturnData
017340
017350      Compute XLeft = (XRight - SWP-WIDTH) / 2
017360      Compute YBOTTOM = (YTop - SWP-HEIGHT) / 2
017370
017380      Call OS2API 'WinSetWindowPos' using
017390          by value hwnd
017400          by value HWND-TOP
017410          by value XLeft
017420          by value YBottom
017430          by value ShortNull
017440          by value ShortNull
017450          by value SWP-DIALOG
017460          returning ReturnData
017470
017480      when WM-COMMAND
017490*
017500* WM-COMMAND carries the user selected pushbutton
017510*
017520      Set Address of DPMCall to PMCallPointer
017530      Move MsgParm1w1 to DPMCRetn
017540      Call OS2API 'WinDismissDlg' using
017550          by value hwnd
017560          by value DPMCRetn
017570
017580      when WM-CONTROL
017590          If MsgParm1w2 = LN-SELECT
017600*
017610* Establish addressability to the Main Program save area
017620*
017630          Set Address of DPMCall to PMCallPointer
017640*
017650* Query dialog for index of user selected item
017660*
```

```

017670          Call OS2API 'WinSendDlgItemMsg' using
017680              by value hwnd
017690              by value ID-PMCallListBox
017700              by value LM-QUERYSELECTION
017710              by value LongNull
017720              by value LongNull
017730          returning ReturnData
017740*
017750* Compute values to retrieve and save call string
017760*
017770          Compute MaxStringLength = DPMCLength - 9
017780          Compute UShortWork = IDS-PMCall1 + ReturnData
017790          Move spaces to DPMCL listBoxEntry
017800          Call OS2API 'WinLoadString' using
017810              by value hab
017820              by value ShortNull
017830              by value UShortWork
017840              by value MaxStringLength
017850              by reference DPMCL listBoxEntry
017860          returning DPMCL listBoxSize
017870*
017880* Pass the chosen PM call back to the Main Program Section.
017890*
017900          Call OS2API 'WinSetDlgItemText' using
017910              by value hwnd
017920              by value ID-PMCallChosen
017930              by reference DPMCL listBoxEntry
017940          returning ReturnData
017950
017960      End-if
017970
017980      If MsgParm1w2 = LN-ENTER
017990*
018000* User double-clicked on item, generate an OK pushbutton message
018010*
018020          Call OS2API 'WinPostMsg' using
018030              by value hwnd
018040              by value WM-COMMAND
018050              by value ID-PMCallButtonOK
018060              by value LongNull
018070          returning ReturnData
018080      End-if
018090
018100      when other
018110          Call OS2API 'WinDefDlgProc' using
018120              by value hwnd
018130              by value Msg
018140              by value MsgParm1
018150              by value MsgParm2
018160          returning Mresult
018170      End-evaluate

```

```
018180      exit program returning Mresult.  
018190*****  
018200* 400-Fill-Box. This perform obtains the call strings from the  
018210*     Resource file and sends them to the dialog's list box.  
018220*****  
018230 400-Fill-Box.  
018240      Compute UShortWork = UShortWork + 1  
018250*  
018260* Load call string from Resource  
018270*  
018280      Compute MaxStringLength = DPMCLength - 9  
018290      Call OS2API 'WinLoadString' using  
018300          by value      hab  
018310          by value      ShortNull  
018320          by value      UShortWork  
018330          by value      MaxStringLength  
018340          by reference DPMCL listBoxEntry  
018350          returning    DPMCL listBoxSize  
018360*  
018370* Insert string into List Box  
018380*  
018390      If DPMCL listBoxSize > 0  
018400          Call OS2API 'WinSendDlgItemMsg' using  
018410              by value      hwnd  
018420              by value      ID-PMCallListBox  
018430              by value      LM-INSERTITEM  
018440              by value      LIT-END  
018450              by reference DPMCL listBoxEntry  
018460              returning    ReturnData  
018470      end-if.  
018480*****  
018490
```

```

018500 WEditDlgProc section.
018510 ****
018520* Window Edit Dialog Procedure: WEditDlgProc.
018530*   This procedure allows the user to set the color of the Main
018540*   window and all child windows plus set the title displayed on
018550*   the Main window. User input is saved in the OS2.INI file.
018560 ****
018570 entry 'WEditDlgProc' using by value hwnd
018580                           by value msg
018590                           by value MsgParm1
018600                           by value MsgParm2.
018610
018620     move 0 to Mresult
018630     evaluate Msg
018640
018650     when WM-INITDLG
018660*
018670* Save the pointer to the PMCall data pass area
018680*
018690     Set WEPointer to Msg2Pointer
018700     Set Address of DWEdit to WEPointer
018710*
018720* Preset the color button to White
018730*
018740     Move 1 to MsgParm1w1
018750     Move 0 to MsgParm1w2 MsgParm2
018760     Call OS2API 'WinSendDlgItemMsg' using
018770           by value hwnd
018780           by value DWE-BN-Color
018790           by value BM-SETCHECK
018800           by value MsgParm1
018810           by value MsgParm2
018820           returning ReturnData
018830*
018840* Preset maximum length of user Main window title input
018850*
018860     Compute MsgParm1w1 = DWELength - 15
018870     Move 0 to MsgParm1w2 MsgParm2
018880     Call OS2API 'WinSendDlgItemMsg' using
018890           by value hwnd
018900           by value ID-WindowName
018910           by value EM-SETTEXTLIMIT
018920           by value MsgParm1
018930           by value MsgParm2
018940           returning ReturnData
018950

```

```
018960      Call OS2API 'WinSendDlgItemMsg' using
018970          by value hwnd
018980          by value ID-WindowName
018990          by value EM-QUERYCHANGED
019000          by value LongNull
019010          by value LongNull
019020          returning ReturnData
019030
019040      Call OS2API 'WinQueryWindowRect' using
019050          by value HWND-DESKTOP
019060          by reference rcl
019070          returning ReturnData
019080
019090      Call OS2API 'WinQueryWindowPos' using
019100          by value hwnd
019110          by reference SWP
019120          returning ReturnData
019130
019140      Compute XLeft = (XRight - SWP-WIDTH) / 2
019150      Compute YBOTTOM = (YTop - SWP-HEIGHT) / 2
019160
019170      Call OS2API 'WinSetWindowPos' using
019180          by value hwnd
019190          by value HWND-TOP
019200          by value XLeft
019210          by value YBottom
019220          by value ShortNull
019230          by value ShortNull
019240          by value SWP-DIALOG
019250          returning ReturnData
019260
019270      when WM-COMMAND
019280          Set Address of DWEdit to WEPointer
019290*
019300* When any pushbutton selected
019310*
019320          Move MsgParm1w1 to DlgReturn
019330*
019340* Check if title text entered
019350*
019360          Call OS2API 'WinSendDlgItemMsg' using
019370          by value hwnd
019380          by value ID-WindowName
019390          by value EM-QUERYCHANGED
019400          by value LongNull
019410          by value LongNull
019420          returning ReturnData
019430
019440          If ReturnTrue
019450*
019460* If text entered obtain window handle and query text
```

```
019470*
019480      Call OS2API 'WinWindowFromID' using
019490          by value hwnd
019500          by value ID-WindowName
019510          returning LongWork
019520
019530      Compute ShortWork = DWELength - 15
019540      Call OS2API 'WinQueryWindowText' using
019550          by value LongWork
019560          by value ShortWork
019570          by reference DWEMainTitle
019580          returning ShortWork
019590
019600      End-if
019610
019620      Call OS2API 'WinDISSmissDlg' using
019630          by value hwnd
019640          by value DlgReturn
019650
019660      when WM-CONTROL
019670          Set Address of DWEdit to WEPointer
019680
019690      If MsgParm1w2 = BN-CLICKED
019700*
019710* If window box checked, determine the checked state of the Box
019720*
019730          If MsgParm1w1 is greater than or equal to
019740              ID-CB-MainWindow
019750
019760          Compute SelectedColor = DWE-BN-Color - 700
019770              If SelectedColor = 0
019780                  Move -2 to SelectedColor
019790              End-if
019800
019810          Call OS2API 'WinSendDlgItemMsg' using
019820              by value hwnd
019830              by value MsgParm1w1
019840              by value BM-QUERYCHECK
019850              by value LongNull
019860              by value LongNull
019870              returning ReturnData
019880*
019890* Set the window selected box to match the current state
019900*
019910          If MsgParm1w1 = ID-CB-MainWindow
019920              If ReturnData = 0
019930                  Move 0 to DWEMWindowColor
019940              Else
019950                  Move SelectedColor to DWEMWindowColor
019960              End-if
019970          Else
```

```
019980           If ReturnData = 0
019990             Move 0 to DWECWindowColor
020000           Else
020010             Move SelectedColor to DWECWindowColor
020020           End-if
020030         End-if
020040       End-if
020050*
020060* If button selected, save window color & calculate text color
020070*
020080           If MsgParm1w1 is
020090             greater than or equal to ID-BN-White
020100               and
020110               MsgParm1w1 is
020120               less than or equal to ID-BN-LT-Gray
020130               Move MsgParm1w1 to DWE-BN-Color
020140               Move CLR-BLACK to DWETextColor
020150
020160             Evaluate MsgParm1w1
020170               when ID-BN-BLUE
020180                 Move CLR-WHITE to DWETextColor
020190               when ID-BN-BROWN
020200                 Move CLR-WHITE to DWETextColor
020210               when ID-BN-DK-BLUE
020220                 Move CLR-WHITE to DWETextColor
020230               when ID-BN-DK-Green
020240                 Move CLR-WHITE to DWETextColor
020250               when ID-BN-DK-Red
020260                 Move CLR-WHITE to DWETextColor
020270               when ID-BN-DK-Gray
020280                 Move CLR-WHITE to DWETextColor
020290             End-evaluate
020300           End-if
020310         End-if
020320
020330       when other
020340         Call OS2API 'WinDefDlgProc' using
020350           by value hwnd
020360           by value Msg
020370           by value MsgParm1
020380           by value MsgParm2
020390           returning Mresult
020400
020410       End-evaluate
020420     exit program returning Mresult.
020430*****
```

```

/***** SAMPLE PROGRAM HEADER FILE (SAMPLE.H) ****/
/*
 * This file defines the C symbolic constants used in the SAMPLE.RC
 * Resource Script file. The equivalent constants are defined to
 * SAMPLE.CBL in the Working Storage Section.
 */
*****



#define ID_MainWind          10
#define ID_File               110
#define MI_New                111
#define MI_Open               112
#define MI_File               113
#define MI_Xtract              114
#define MI_Print              115
#define MI_Exit                116
#define ID_Edit                120
#define MI_Clear              121
#define MI_Copy                122
#define MI_Wedit              123
#define ID_View                130
#define MI_Object              131
#define MI_Source              132
#define ID_Option              140
#define MI_Msg                 141
#define ID_Window              150
#define MI_Cascade              151
#define MI_Titled              152
#define ID_Help                160
#define MI_Xhelp              161
#define MI_About              162

#define ID_SourceWind          20
#define ID_Close               210
#define MI_Closesw              211
#define MI_Resumesw              212

#define IDS_PMCall1           500
#define IDS_PMCall2           501
#define IDS_PMCall3           502
#define IDS_PMCall4           503
#define IDS_PMCall5           504
#define IDS_PMCall6           505
#define IDS_PMCall7           506
#define IDS_PMCall8           507
#define IDS_PMCall9           508
#define IDS_PMCall10          509

```

```
#define IDS_PMCall111      510
#define IDS_PMCall112      511
#define IDS_PMCall113      512
#define IDS_PMCall114      513
#define IDS_PMCall115      514
#define IDS_PMCall116      515
#define IDS_PMCall117      516
#define IDS_PMCall118      517
#define IDS_PMCall119      518
#define IDS_PMCall120      519
#define IDS_PMCall121      520
#define IDS_PMCall122      521
#define IDS_PMCall123      522
#define IDS_PMCall124      523
#define IDS_PMCall125      524
#define IDS_PMCall126      525
#define IDS_PMCall127      526
#define IDS_PMCall128      527
#define IDS_PMCall129      528
#define IDS_PMCall130      529
#define IDS_PMCall131      530
#define IDS_PMCall132      531
#define IDS_PMCall133      532
#define IDS_PMCall134      533
#define IDS_PMCall135      534
#define IDS_PMCall136      535
#define IDS_PMCall137      536
#define IDS_PMCall138      537
#define IDS_PMCall139      538
#define IDS_PMCall140      539
#define IDS_PMCall141      540
#define IDS_PMCall142      541

#define IDS_MsgBoxOne       801
#define IDS_MsgBoxTwo       802
#define ID_Handptr          901
#define ID_LHandptr         902
#define ID_DHandptr         903
```

```

***** */
/*
/*          The COBOL/2 Sample Resource Script File (.RC) */
/*
/*****
/*
/* The resources used by the COBOL/2 SAMPLE.CBL program are defined */
/* in this Resource Script file . This file is compiled by the */
/* Resource compiler as part of the SAMPLE program compilation and */
/* link process to produce an executable-format file. */
/*
/* In this Resource Script file for ID_MAINWIND: */
/-----
/*
/* ACCELTABLE      Key combinations for main window menu */
/* RCINCLUDE        Pointer to dialog box templates */
/* ICON             Sample program window Icon */
/* MENU            Sample program main window menu */
/* POINTER          Sample program client window pointer */
/* STRINGTABLE     Sample program text strings */
/*
/*****
*/
#include "os2.h"
#include "sample.h"
#include "termdlg.h"
#include "pmcall.h"
#include "winedit.h"

ICON ID_MainWind SAMPLE.ICO

POINTER ID_Handptr HAND.PTR
POINTER ID_LHandptr LHAND.PTR
POINTER ID_DHandptr DHAND.PTR

ACCELTABLE ID_MainWind
BEGIN
    "a",MI_About,CONTROL
    "e",MI_Exit,CONTROL
    "s",MI_Source,CONTROL
    "w",MI_Wedit,CONTROL
END

ACCELTABLE ID_SourceWind
BEGIN
    "l",MI_Closesw,CONTROL
END

STRINGTABLE PRELOAD
BEGIN

```

```
IDS_MsgBoxOne, "This is the COBOL PM sample program"
IDS_MsgBoxTwo, "Install the Courier font before running the Sample program."
END

STRINGTABLE
BEGIN
  IDS_PMCall1,   "GpiCharStringAt"
  IDS_PMCall2,   "GpiCreateLogFont"
  IDS_PMCall3,   "GpiDeleteSetId"
  IDS_PMCall4,   "GpiErase"
  IDS_PMCall5,   "GpiQueryFontFileDescriptions"
  IDS_PMCall6,   "GpiQueryFontMetrics"
  IDS_PMCall7,   "GpiSetCharSet"
  IDS_PMCall8,   "WinBeginEnumWindows"
  IDS_PMCall9,   "WinBeginPaint"
  IDS_PMCall10,  "WinCreateMsgQueue"
  IDS_PMCall11,  "WinCreateStdWindow"
  IDS_PMCall12,  "WinDefDlgProc"
  IDS_PMCall13,  "WinDefWindowProc"
  IDS_PMCall14,  "WinDestroyMsgQueue"
  IDS_PMCall15,  "WinDestroyWindow"
  IDS_PMCall16,  "WinDismissDlg"
  IDS_PMCall17,  "WinDispatchMsg"
  IDS_PMCall18,  "WinDlgBox"
  IDS_PMCall19,  "WinEndEnumWindows"
  IDS_PMCall20,  "WinEndPaint"
  IDS_PMCall21,  "WinFillRect"
  IDS_PMCall22,  "WinGetMsg"
  IDS_PMCall23,  "WinGetNextWindow"
  IDS_PMCall24,  "WinInitialize"
  IDS_PMCall25,  "WinInvalidateRegion"
  IDS_PMCall26,  "WinLoadPointer"
  IDS_PMCall27,  "WinLoadString"
  IDS_PMCall28,  "WinMessageBox"
  IDS_PMCall29,  "WinPostMsg"
  IDS_PMCall30,  "WinQueryWindowRect"
  IDS_PMCall31,  "WinQuerySysValue"
  IDS_PMCall32,  "WinQueryWindowPos"
  IDS_PMCall33,  "WinRegisterClass"
  IDS_PMCall34,  "WinSendDlgItemMsg"
  IDS_PMCall35,  "WinSendMsg"
  IDS_PMCall36,  "WinSetDlgItemText"
  IDS_PMCall37,  "WinSetPointer"
  IDS_PMCall38,  "WinSetWindowPosition"
  IDS_PMCall39,  "WinSetWindowText"
  IDS_PMCall40,  "WinShowWindow"
  IDS_PMCall41,  "WinTerminate"
  IDS_PMCall42,  "WinWindowFromID"
END

MENU ID_MainWind PRELOAD
```

```
BEGIN
    SUBMENU  "~File",      ID_File
    BEGIN
        MENUITEM  "~New",          MI_New,,   MIA_DISABLED
        MENUITEM  "~Open...",     MI_Open,,  MIA_DISABLED
        MENUITEM  "~File...",     MI_File,, MIA_DISABLED
        MENUITEM  "~Xtract...",   MI_Xtract,,MIA_DISABLED
        MENUITEM  SEPARATOR
        MENUITEM  "~Print...",   MI_Print,, MIA_DISABLED
        MENUITEM  SEPARATOR
        MENUITEM  "~Exit",       MI_Exit
    END
    SUBMENU  "~Edit",      ID_Edit
    BEGIN
        MENUITEM  "C~lear",       MI_Clear,, MIA_DISABLED
        MENUITEM  "~Copy...",    MI_Copy,,  MIA_DISABLED
        MENUITEM  "~Window Edit...\\tCtrl+w", MI_Wedit
    END
    SUBMENU  "~View",       ID_View
    BEGIN
        MENUITEM  "~Object",     MI_Object,,MIA_DISABLED
        MENUITEM  "~Source\\tCtrl+s", MI_Source
    END
    SUBMENU  "~Options",   ID_Option
    BEGIN
        MENUITEM  "~Messages",   MI_Msg,,   MIA_DISABLED
    END
    SUBMENU  "~Window",    ID_Window
    BEGIN
        MENUITEM  "~Cascade",    MI_Cascade,,MIA_DISABLED
        MENUITEM  "~Titled",     MI_Titled,,MIA_DISABLED
    END
    SUBMENU  "~Help",      ID_Help
    BEGIN
        MENUITEM  "E~xtended Help...", MI_Xhelp,,MIA_DISABLED
        MENUITEM  "~About\\tCtrl+a", MI_About
    END
END
MENU      ID_SourceWind
BEGIN
    SUBMENU  "~Close",      ID_Close
    BEGIN
        MENUITEM  "C~lose\\tCtrl+l",   MI_Closesw
        MENUITEM  "~Resume",       MI_Resumesw
    END
END
RCINCLUDE TERMDLG.DLG
RCINCLUDE PMCALL.DLG
RCINCLUDE WINEDIT.DLG
```

```
*****/*  
/*  
/*      TERMINATION DIALOG HEADER FILE (TERMDLG.H)  
/*  
/*  
*****/*  
/*  
/* This file defines the C symbolic constants used in the  
/* TermDlgBox Dialog file. Equivalent constants that are referenced */  
/* are declared in the program's working storage section.  
/*  
*****/*  
  
#define ID_TermDlgBox          700  
#define ID_TermOK              701  
#define ID_TermNo              702  
#define ID_TermTxt1            703  
#define ID_TermTxt2            704
```

```
DLGTEMPLATE ID_TermDlgBox
BEGIN
    DIALOG "Sample Program", ID_TermDlgBox,
        66, 42, 204, 57,
        WS_CLIPSIBLINGS | WS_SAVEBITS,
        FCF_TITLEBAR | FCF_NOBYTEALIGN |
        FCF_DLGBORDER
    BEGIN
        CONTROL "OK", ID_TermOK,
            14, 5, 38, 13,
            WC_BUTTON, BS_PUSHBUTTON |
            BS_DEFAULT | WS_TABSTOP | WS_VISIBLE
        CONTROL "No", ID_TermNo,
            62, 5, 38, 13,
            WC_BUTTON, BS_PUSHBUTTON |
            WS_TABSTOP | WS_VISIBLE
        CONTROL "The sample program is about to end.", ID_TermTxt1,
            1, 35, 200, 8,
            WC_STATIC, SS_TEXT | DT_CENTER |
            DT_VCENTER | WS_GROUP | WS_VISIBLE
        CONTROL "OK to end?", ID_TermTxt2,
            1, 25, 200, 8,
            WC_STATIC, SS_TEXT | DT_CENTER |
            DT_VCENTER | WS_GROUP | WS_VISIBLE
    END
END
```

```
*****  
/*  
/*      PM CALL SPECIFICATION DIALOG HEADER FILE (PMCALL.H)  
/*  
*****  
/*  
/* This file defines the C symbolic constants used in the  
/* PMCALL Dialog file.  The equivalent constants are defined  
/* to SAMPLE.CBL in the SAMPLE.CIN COBOL include file.  
/*  
*****  
  
#define ID_PMCallDlgBox          705  
#define ID_PMCallListBox         706  
#define ID_PMCallButtonCan       707  
#define ID_PMCallButtonOK        708  
#define ID_PMCallText1           709  
#define ID_PMCallChosen          710  
#define ID_PMCallText2           711  
#define ID_PMCallText3           712  
#define ID_PMCallText4           713  
#define ID_PMCallGpBox           714
```

```
DLGTEMPLATE ID_PMCALLDlgBox
BEGIN
    DIALOG "Select A PM Call", ID_PMCALLDlgBox,
        92, 86, 230, 96,
        FS_NOBYTEALIGN | FS_DLGBORDER | FS_BORDER |
        WS_CLIPSIBLINGS | WS_SAVEBITS,
        FCF_TITLEBAR
    BEGIN
        CONTROL "", ID_PMCALLListBox,
            138, 8, 82, 86,
            WC_LISTBOX,
            WS_TABSTOP | WS_VISIBLE
        CONTROL "OK", ID_PMCALLButtonOK,
            9, 5, 49, 13, WC_BUTTON,
            BS_PUSHBUTTON | BS_DEFAULT |
            WS_TABSTOP | WS_VISIBLE
        CONTROL "Cancel", ID_PMCALLButtonCan,
            66, 5, 49, 13,
            WC_BUTTON,
            BS_PUSHBUTTON |
            WS_TABSTOP | WS_VISIBLE
        CONTROL "PM Call:", ID_PMCALLText1,
            9, 80, 40, 8,
            WC_STATIC,
            SS_TEXT | DT_LEFT | DT_VCENTER |
            WS_GROUP | WS_VISIBLE
        CONTROL "", ID_PMCALLChosen,
            5, 70, 124, 8,
            WC_STATIC,
            SS_TEXT | DT_LEFT | DT_VCENTER |
            WS_GROUP | WS_VISIBLE
        CONTROL "", ID_PMCALLGpBox,
            3, 68, 129, 15,
            WC_STATIC,
            SS_GROUPBOX |
            WS_GROUP | WS_VISIBLE
        CONTROL "Select a PM call from the", ID_PMCALLText2,
            8, 52, 110, 8,
            WC_STATIC,
            SS_TEXT | DT_LEFT | DT_VCENTER |
            WS_GROUP | WS_VISIBLE
        CONTROL "list at the right then press", ID_PMCALLText3,
            9, 45, 115, 8,
            WC_STATIC,
            SS_TEXT | DT_LEFT | DT_VCENTER |
            WS_GROUP | WS_VISIBLE
        CONTROL "the OK button", ID_PMCALLText4,
            8, 38, 58, 8,
            WC_STATIC,
            SS_TEXT | DT_LEFT | DT_VCENTER |
            WS_GROUP | WS_VISIBLE
    END
END
```

```
*****  
/* *****  
* Window Edit Dialog Header File (WinEdit.H) *  
* *****  
*  
/* This file contains the C symbolic constants used in the WinEditDlg **/  
/* dialog file. Equivalent constants that are referenced are declared**/  
/* in the program's Working Storage Section. **/  
/* *****  
  
#define ID_BN_White          700  
#define ID_BN_Blue           701  
#define ID_BN_Red            702  
#define ID_BN_Green          704  
#define ID_BN_Cyan           705  
#define ID_BN_Yellow         706  
#define ID_BN_DK_Gray        708  
#define ID_BN_DK_Blue        709  
#define ID_BN_DK_Red         710  
#define ID_BN_DK_Green       712  
#define ID_BN_Brown          714  
#define ID_BN_LT_Gray        715  
#define ID_WinEditDlgBox     716  
#define ID_WinEditGpBox1     717  
#define ID_WinEditGpBox2     718  
#define ID_CB_MainWindow     719  
#define ID_CB_ChildWindow    720  
#define ID_WindowName        721  
#define ID_WinDlgText1       722  
#define ID_WinDlgButtonOK     723  
#define ID_WinDlgButtonCan    724
```

```
DLGTEMPLATE ID_WinEditDlgBox LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG "Window Edit", ID_WinEditDlgBox,
        33, 64, 373, 148,
        WS_CLIPSIBLINGS | WS_SAVEBITS,
        FCF_TITLEBAR | FCF_DLGBORDER |
        FCF_NOBYTEALIGN
    BEGIN
        CONTROL "Window Colors", ID_WinEditGpBox1,
            18, 53, 183, 89,
            WC_STATIC, SS_GROUPBOX |
            WS_VISIBLE
        CONTROL "Blue", ID_BN_Blue,
            32, 119, 58, 10,
            WC_BUTTON, BS_AUTORADIOBUTTON |
            WS_VISIBLE | WS_GROUP
        CONTROL "Brown", ID_BN_Brown,
            32, 108, 50, 10,
            WC_BUTTON, BS_AUTORADIOBUTTON |
            WS_VISIBLE
        CONTROL "Cyan", ID_BN_Cyan,
            32, 96, 39, 10,
            WC_BUTTON, BS_AUTORADIOBUTTON |
            WS_VISIBLE
        CONTROL "Dark Blue", ID_BN_DK_Blue,
            32, 84, 57, 10,
            WC_BUTTON, BS_AUTORADIOBUTTON |
            WS_VISIBLE
        CONTROL "Dark Green", ID_BN_DK_Green,
            32, 72, 62, 10,
            WC_BUTTON, BS_AUTORADIOBUTTON |
            WS_VISIBLE
        CONTROL "Dark Red", ID_BN_DK_Red,
            32, 60, 55, 10,
            WC_BUTTON, BS_AUTORADIOBUTTON |
            WS_VISIBLE
        CONTROL "Dark Gray", ID_BN_DK_Gray,
            115, 120, 58, 10,
            WC_BUTTON, BS_AUTORADIOBUTTON |
            WS_VISIBLE
        CONTROL "Green", ID_BN_Green,
            115, 108, 48, 10,
            WC_BUTTON, BS_AUTORADIOBUTTON |
            WS_VISIBLE
        CONTROL "Light Gray", ID_BN_LT_Gray,
            115, 96, 62, 10,
            WC_BUTTON, BS_AUTORADIOBUTTON |
            WS_VISIBLE
        CONTROL "Red", ID_BN_Red,
            115, 84, 40, 10,
            WC_BUTTON, BS_AUTORADIOBUTTON |
```

```
          WS_VISIBLE
CONTROL "Yellow", ID_BN_Yellow,
        115, 72, 48, 10,
        WC_BUTTON, BS_AUTORADIOBUTTON |
        WS_VISIBLE
CONTROL "White", ID_BN_White,
        115, 60, 56, 10,
        WC_BUTTON, BS_AUTORADIOBUTTON |
        WS_VISIBLE
CONTROL "Window", ID_WinEditGpBox2,
        218, 53, 136, 88,
        WC_STATIC, SS_GROUPBOX |
        WS_VISIBLE
CONTROL "Main Window", ID_CB_MainWindow,
        250, 102, 79, 10,
        WC_BUTTON, BS_AUTOCHECKBOX |
        WS_TABSTOP | WS_VISIBLE | WS_GROUP
CONTROL "Child Windows", ID_CB_ChildWindow,
        250, 83, 84, 10,
        WC_BUTTON, BS_AUTOCHECKBOX |
        WS_TABSTOP | WS_VISIBLE
CONTROL "", ID_WindowName,
        78, 28, 220, 8,
        WC_ENTRYFIELD, ES_LEFT | ES_MARGIN |
        WS_GROUP | WS_TABSTOP | WS_VISIBLE
CONTROL "Main Window Title", ID_WinDlgText1,
        1, 41, 365, 8,
        WC_STATIC, SS_TEXT |
        DT_CENTER | DT_TOP | WS_GROUP | WS_VISIBLE
CONTROL "OK", ID_WinDlgButtonOK,
        85, 6, 64, 13,
        WC_BUTTON, BS_PUSHBUTTON | WS_TABSTOP |
        WS_GROUP | WS_VISIBLE
CONTROL "Cancel", ID_WinDlgButtonCan,
        229, 6, 64, 13,
        WC_BUTTON, BS_PUSHBUTTON |
        WS_GROUP | WS_TABSTOP | WS_VISIBLE
END
END
```

```
000010 IDENTIFICATION DIVISION.  
000020 PROGRAM-ID. COBOL-PM-PRT-SETUP.  
000030 DATE-COMPILED. 01-02-92.  
000040 AUTHOR. DAVID DILL.  
000050  
000060 ENVIRONMENT DIVISION.  
000070 CONFIGURATION SECTION.  
000080 SOURCE-COMPUTER. IBM-PERSONAL-SYSTEM-2.  
000090 OBJECT-COMPUTER. IBM-PERSONAL-SYSTEM-2.  
000100  
000110 SPECIAL-NAMES.  
000120      call-convention 3 is OS2API.  
000130  
000140 WORKING-STORAGE SECTION.  
000150*  
000160* Window handles  
000170*  
000180 77 hwndClient          pic s9(9) comp-5.  
000190 77 hwndSubWindowQuery  pic s9(9) comp-5.  
000200 77 HINI-SYSTEMPROFILE pic s9(9) comp-5 value -2.  
000210 77 HWND-DESKTOP       pic s9(9) comp-5 value 1.  
000220 77 hwndDLL           pic 9(4) comp-5.  
000230*  
000240* PM Message IDs  
000250*  
000260 77 WM-COMMAND        pic 9(4) comp-5 value 32.  
000270 77 WM-CONTROL         pic 9(4) comp-5 value 48.  
000280 77 WM-INITDLG         pic 9(4) comp-5 value 59.  
000290*  
000300* List Box Commands and Messages  
000310*  
000320 77 LIT-END            pic s9(9) comp-5 value -1.  
000330 77 LN-SELECT           pic 9(4) comp-5 value 1.  
000340 77 LN-ENTER            pic 9(4) comp-5 value 5.  
000350 77 LM-INSERTITEM       pic 9(4) comp-5 value 353.  
000360 77 LM-SELECTITEM       pic 9(4) comp-5 value 356.  
000370 77 LM-QUERYSELECTION   pic 9(4) comp-5 value 357.  
000380 77 LM-DELETEALL        pic 9(4) comp-5 value 366.  
000390*  
000400* Entry Field Commands and Messages  
000410*  
000420 77 EM-SETTEXTLIMIT     pic 9(4) comp-5 value 323.  
000430*  
000440* String Storage  
000450*  
000460 77 MaxStringLength     pic 9(4) comp-5 value 0.  
000470*  
000480* INI application name  
000490*  
000500 01 PMQName            pic x(19)  
                                value "PM_SPOOLER_PRINTER" & x"00".  
000510
```

```
000520*
000530* Location of printer setup dialog for Version 1.3
000540*
000550*01 DLLName           pic x(24)
000560*                     value "C:\OS2\DLL\PRTDLGBX.DLL" & x"00".
000570*
000580* Location of printer setup dialog for Version 2.0
000590*
000600 01 DLLName           pic x(29)
000610                     value "C:\OS2\APPS\DLL\PRTDLGBX.DLL" & x"00".
000620*
000630* Resource File Definitions
000640*
000650 77 ID-PrinterDlgBox      pic 9(4) comp-5 value 740.
000660 77 ID-QueueListBox       pic 9(4) comp-5 value 742.
000670 77 ID-DriverListBox      pic 9(4) comp-5 value 744.
000680 77 ID-PrinterQueue       pic 9(4) comp-5 value 746.
000690 77 ID-PrinterDriver      pic 9(4) comp-5 value 749.
000700 77 ID-PrinterCopies      pic 9(4) comp-5 value 752.
000710 77 ID-PrinterButtonOK     pic 9(9) comp-5 value 753.
000720 77 ID-PrinterButtonCan    pic 9(9) comp-5 value 754.
000730*
000740* Miscellaneous Definitions
000750*
000760 77 ReturnData          pic s9(4) comp-5.
000770     88 ReturnTrue         value 1.
000780     88 ReturnFalse        value 0.
000790 77 SemiColon           pic x value ";".
000800 77 Period              pic x value "..".
000810 77 ValueComma          pic x value ",".
000820 77 DlgReturn           pic 9(4) comp-5.
000830 77 ShortWork           pic s9(4) comp-5.
000840 77 UShortWork          pic 9(4) comp-5.
000850 77 ULONGWork           pic 9(9) comp-5.
000860 77 LongNull            pic 9(9) comp-5 value 0.
000870 77 ShortNull           pic 9(4) comp-5 value 0.
000880 77 Indx                pic 9(4) comp-5.
000890 77 KeyName             pic x(35) value spaces.
000900 77 RetLength            pic 9(9) comp-5 value 0.
000910 77 ParmCount           pic 9(4) comp-5 value 0.
000920 77 NullString          pic x value x"00".
000930 77 OneString            pic xx value x"3100".
000940*
000950* Storage arrays
000960*
000970 01 ProfileString        pic x(200).
000980 01 PrtString.
000990   05 PrtStringWork       pic x occurs 200 times.
001000*
001010* Array of logical printers
001020*
```

```
001030 01 PMLPrinters.
001040      05 LPrinters          pic x(33) occurs 5 times.
001050*
001060* Array of print queue names
001070*
001080 01 PMQueues.
001090      05 QueueNames        pic x(13) occurs 5 times.
001100*
001110* Array of all printer drivers per queue
001120*
001130 01 PMDrivers.
001140      05 PrtDrivers        pic x(150) occurs 5 times.
001150*
001160* Array of individual print drivers for selected queue
001170*
001180 01 PMPrintDrivers.
001190      05 PrintDrivers       pic x(35) occurs 5 times.
001200*
001210* Printer related counters
001220*
001230 77 LPCount              pic s9(4) comp-5 value 0.
001240 77 UNSPtr               pic 9(4) comp-5 value 0.
001250 77 StopByte             pic x.
001260 77 QueueCount           pic s9(4) comp-5 value 0.
001270 77 DriverCount          pic s9(4) comp-5 value 0.
001280*
001290* Selected Printer queue, driver and divided driver name
001300*
001310*
001320* Pointer to the PMPrint passed data area
001330*
001340 77 PMDataPointer        usage is pointer.
001350*
001360* COBOL Procedure pointers
001370*
001380 77 WindowProc           procedure-pointer.
001390
001400 LOCAL-STORAGE SECTION.
001410 01 Mresult              pic x(4) comp-5.
001420
001430 LINKAGE SECTION.
001440 01 hwnd                 pic 9(9) comp-5.
001450 01 Msg                  pic 9(4) comp-5.
001460 01 MsgParm1             pic 9(9) comp-5.
001470 01 Redefines MsgParm1.
001480      05 MsgParm1w1        pic 9(4) comp-5.
001490      05 MsgParm1w2        pic 9(4) comp-5.
001500 01 MsgParm2              pic 9(9) comp-5.
001510 01 Redefines MsgParm2.
001520      05 MsgParm2w1        pic 9(4) comp-5.
001530      05 MsgParm2w2        pic 9(4) comp-5.
```

402 The COBOL Presentation Manager Programming Guide

```
001540 01 Msg2Pointer           usage is pointer redefines Msgparm2.
001550 01 MsgTime              pic 9(9) comp-5.
001560 01 MsgXPoint            pic 9(9) comp-5.
001570 01 MsgYPoint            pic 9(9) comp-5.
001580*
001590* PMPrint dialog data pass structure (CreateParms)
001600*
001610 01 DPMPrint.
001620    05 DPMPLength          pic 9(4) comp-5.
001630    05 DPMPType            pic 9(4) comp-5.
001640    05 DPMPRtCopies        pic X(3).
001650    05 DPMPQueueName       pic x(20).
001660    05 DPMPDriverName      pic x(40).
001670    05 DPMPDrvrvName       pic x(10).
001680    05 DPMPPrtName         pic x(30).
001690
001700 PROCEDURE DIVISION OS2API.
001710
```

```
001720 PRTSAMPL.  
001730*****  
001740  
001750*****  
001760* Printer Dialog Procedure: PrintDlgProc.  
001770* This procedure determines which of the available queues and  
001780* drivers the user wishes to use for output.  
001790*****  
001800 entry 'Prtsampl' using by value MsgParm1  
001810           by value MsgParm2  
001820  
001830     Move MsgParm1 to hwndClient  
001840     Set PMDataPointer to Msg2Pointer  
001850     Set Address of DPMPrint to PMDataPointer  
001860  
001870     Call OS2API 'DosLoadModule' using  
001880           by reference profilestring  
001890           by value    200 size 2  
001900           by reference DLLName  
001910           by reference hwndDLL  
001920           returning   ReturnData  
001930  
001940     Set WindowProc to ENTRY 'PrintDlgProc'  
001950     Call OS2API 'WinDlgBox' using  
001960           by value  HWND-DESKTOP  
001970           by value  hwndClient  
001980           by value  WindowProc  
001990           by value  hwndDLL  
002000           by value  ID-PrinterDlgBox  
002010           by value  LongNull  
002020           returning DlgReturn  
002030  
002040     Move DlgReturn to Mresult.  
002050     exit program returning Mresult.  
002060*****  
002070  
002080 PrintDlgProc Section.  
002090*****  
002100* Printer Dialog Procedure: PrintDlgProc.  
002110* This procedure determines which of the available queues and  
002120* drivers the user wishes to use for output.  
002130*****  
002140 entry 'PrintDlgProc' using by value hwnd  
002150           by value msg  
002160           by value MsgParm1  
002170           by value MsgParm2.  
002180*  
002190* Set return code to OK and evaluate the message  
002200*  
002210     move 0 to Mresult  
002220     evaluate Msg
```

```
002230
002240      when WM-INITDLG
002250*
002260* Save the pointer to the PMPrint data pass area
002270*
002280      Set Address of DPMPrint to PMDataPointer
002290*
002300* Extract current printer queue names from OS2SYS.INI
002310*
002320      Call OS2API 'PrfQueryProfileString' using
002330          by value      HINI-SYSTEMPROFILE
002340          by reference PMQName
002350          by value      LongNull
002360          by value      LongNull
002370          by reference ProfileString
002380          by value      50 size 4
002390          returning    RetLength
002400
002410      If RetLength > 0
002420
002430      Move 1 to UNSPtr
002440*
002450* Extract individual logical printer names.
002460*
002470      Perform 600-Extract-LPrinter with test after
002480          varying LPCCount from 1 by 1
002490          until ParmCount = 0
002500
002510      Subtract 1 from LPCCount giving LPCCount
002520*
002530* Extract the printer driver and queue names for each printer.
002540*
002550      Perform 610-Ext-LPStrings with test after
002560          varying QueueCount from 1 by 1
002570          until QueueCount = LPCCount
002580*
002590* Clear the Queue list box
002600*
002610      Call OS2API 'WinSendDlgItemMsg' using
002620          by value      hwnd
002630          by value      ID-QueueListBox
002640          by value      LM-DELETEALL
002650          by value      LongNull
002660          by value      LongNull
002670          returning    ReturnData
002680*
002690* Send queue names to ID-QueueListBox
002700*
002710      Perform 620-Fill-Box with test after
002720          varying Indx from 1 by 1
002730          until Indx = QueueCount
```

```

002740*
002750* Preselect the 1st item in the list as the default
002760*
002770            Call OS2API 'WinSendDlgItemMsg' using
002780                by value    hwnd
002790                by value    ID-QueueListBox
002800                by value    LM-SELECTITEM
002810                by value    LongNull
002820                by value    1 size 4
002830            returning ReturnData
002840*
002850* Set maximum number of copies to 99
002860*
002870            Call OS2API 'WinSendDlgItemMsg' using
002880                by value    hwnd
002890                by value    ID-PrinterCopies
002900                by value    EM-SETTEXTLIMIT
002910                by value    2 size 4
002920                by value    LongNull
002930            returning ReturnData
002940*
002950* Set initial number of copies to 1
002960*
002970            Call OS2API 'WinSetDlgItemText' using
002980                by value    hwnd
002990                by value    ID-PrinterCopies
003000                by reference OneString
003010            returning    ReturnData
003020        Else
003030*
003040* End the dialog when no printers are available
003050*
003060            Move 99 to DlgReturn
003070            Call OS2API 'WinDismissDlg' using
003080                by value    hwnd
003090                by value    DlgReturn
003100        End-if
003110
003120        when WM-COMMAND
003130            Set Address of DPMPrint to PMDataPointer
003140            If MsgParm1w1 = ID-PrinterButtonOK
003150*
003160* Get the handle of the PrinterCopies entry box
003170*
003180            Call OS2API 'WinWindowFromID' using
003190                by value    hwnd
003200                by value    ID-PrinterCopies
003210            returning hwndSubWindowQuery
003220*
003230* Retrieve number of copies
003240*

```

```
003250      Move 3 to MaxStringLength
003260      Call OS2API 'WinQueryWindowText' using
003270          by value      hwndSubWindowQuery
003280          by value      MaxStringLength
003290          by reference DPMPrtCopies
003300          returning     ReturnData
003310*
003320* Break the printer driver name into DriverName and DeviceName
003330*
003340      Move 1 to UShortWork
003350      Unstring DPMPDriverName Delimited by Period or
003360          NullString into DPMPDrvrvName
003370          Delimiter in StopByte
003380          with Pointer UShortWork
003390      End-Unstring
003400      Inspect DPMPDrvrvName replacing
003410          first space by x"00"
003420      Move NullString to DPMPPrtName
003430      If StopByte = Period
003440          Unstring DPMPDriverName Delimited by
003450              NullString into DPMPPrtName
003460              with Pointer UShortWork
003470          End-Unstring
003480      End-if
003490      End-if
003500*
003510* Free the PrinterDlgBox resource DLL
003520*
003530      Call OS2API 'DosFreeModule' using
003540          by value      hwndDLL
003550          returning     ReturnData
003560*
003570* Return button and dismiss the dialog
003580*
003590      Move MsgParm1w1 to DlgReturn
003600      Call OS2API 'WinDismissDlg' using
003610          by value      hwnd
003620          by value      DlgReturn
003630
003640      when WM-CONTROL
003650          Set Address of DPMPrint to PMDataPointer
003660          If MsgParm1w2 = LN-SELECT
003670*
003680* Determine which list box was selected by the user
003690*
003700      Call OS2API 'WinWindowFromID' using
003710          by value      hwnd
003720          by value      ID-QueueListBox
003730          returning     hwndSubWindowQuery
003740
003750      If MsgParm2 = hwndSubWindowQuery
```

```
003760
003770          Call OS2API 'WinSendDlgItemMsg' using
003780                  by value hwnd
003790                  by value ID-QueueListBox
003800                  by value LM-QUERYSELECTION
003810                  by value LongNull
003820                  by value LongNull
003830                  returning Indx
003840
003850          Add 1 to Indx giving Indx
003860          Move QueueNames(Indx) to DPMPQueueName
003870          Call OS2API 'WinSetDlgItemText' using
003880                  by value hwnd
003890                  by value ID-PrinterQueue
003900                  by reference DPMPQueueName
003910                  returning ReturnData
003920*
003930* Extract defined drivers for the selected queue
003940*
003950          Move 1 to UNSPtr
003960          Perform 630-Extract-Drivers With test after
003970                  varying DriverCount from 1 by 1
003980                  until StopByte = NullString
003990*
004000* Clear the Driver list box
004010*
004020          Call OS2API 'WinSendDlgItemMsg' using
004030                  by value hwnd
004040                  by value ID-DriverListBox
004050                  by value LM-DELETEALL
004060                  by value LongNull
004070                  by value LongNull
004080                  returning ReturnData
004090*
004100* Send driver names to ID-DriverListBox
004110*
004120          Perform 625-Fill-Box with test after
004130                  varying Indx from 1 by 1
004140                  until Indx = DriverCount
004150*
004160* Preselect the 1st item in the list as the default
004170*
004180          Call OS2API 'WinSendDlgItemMsg' using
004190                  by value hwnd
004200                  by value ID-DriverListBox
004210                  by value LM-SELECTITEM
004220                  by value LongNull
004230                  by value 1 size 4
004240                  returning ReturnData
004250
004260          Else
```

```
004270          Call OS2API 'WinSendDlgItemMsg' using
004280                  by value hwnd
004290                  by value ID-DriverListBox
004300                  by value LM-QUERYSELECTION
004310                  by value LongNull
004320                  by value LongNull
004330                  returning Indx
004340
004350          Add 1 to Indx giving Indx
004360          Move PrintDrivers(Indx) to DPMPDriverName
004370          Call OS2API 'WinSetDlgItemText' using
004380                  by value hwnd
004390                  by value ID-PrinterDriver
004400                  by reference DPMPDriverName
004410                  returning ReturnData
004420          End-if
004430      End-if
004440
004450      If MsgParm1w2 = LN-ENTER
004460          Call OS2API 'WinPostMsg' using
004470                  by value hwnd
004480                  by value WM-COMMAND
004490                  by value ID-PrinterButtonOK
004500                  by value LongNull
004510                  returning ReturnData
004520      End-if
004530
004540      when other
004550          Call OS2API 'WinDefDlgProc' using
004560                  by value hwnd
004570                  by value Msg
004580                  by value MsgParm1
004590                  by value MsgParm2
004600                  returning Mresult
004610
004620      End-evaluate
004630      exit program returning Mresult.
004640*****
004650* 600-Extract-LPrinter. This call extracts the user defined
004660*     logical printers from the string returned by the
004670*     OS2SYS.INI query. The logical printers are loaded into
004680*     the LPrinters array.
004690*****
004700 600-Extract-LPrinter.
004710      Unstring ProfileString Delimited by NullString
004720          into LPrinters(LPCount) count in ParmCount
004730          with Pointer UNSPtr
004740      End-Unstring
004750      If ParmCount > 0
004760          Inspect LPrinters(LPCount) replacing first space by x"00"
004770      End-if.
```

```
004780
004790*****
004800* 610-Ext-LPStrings. This routine divides the string returned by
004810*      PM_SPOOL_PRINTER into a printer driver string and a queue
004820*      name string. The port string is ignored.
004830*
004840*      Each PM_SPOOL_PRINTER string has the following form.
004850*
004860*          PORT;DRIVER1,DRIVER2,DRIVER3,...;QUEUE;;0
004870*****
004880 610-Ext-LPStrings.
004890      Move LPrinters(QueueCount) to Keyname
004900      Call OS2API 'PrfQueryProfileString' using
004910          by value      HINI-SYSTEMPROFILE
004920          by reference PMQName
004930          by reference Keyname
004940          by value      LongNull
004950          by reference ProfileString
004960          by value      150 size 4
004970          returning     RetLength
004980*
004990* Skip pointer over port name
005000*
005010      Move 0 to UNSPtr
005020      Inspect ProfileString tallying UNSPtr for characters before
005030          initial Semicolon
005040      Add 2 to UNSPtr giving UNSPtr
005050*
005060* Extract the printer drivers string
005070*
005080      Move UNSPtr to ShortWork
005090      Unstring ProfileString Delimited by Semicolon
005100          into PrtString with Pointer UNSPtr
005110      End-Unstring
005120      Subtract ShortWork from UNSPtr giving ShortWork
005130      Move NullString to PrtStringWork(ShortWork)
005140      Move PrtString to PrtDrivers(QueueCount)
005150*
005160* Extract the queue name
005170*
005180      Move UNSPtr to ShortWork
005190      Unstring ProfileString Delimited by Semicolon
005200          into PrtString with Pointer UNSPtr
005210      End-Unstring
005220      Subtract ShortWork from UNSPtr giving ShortWork
005230      Move NullString to PrtStringWork(ShortWork)
005240      Move PrtString to QueueNames(QueueCount).
005250
005260*****
005270* 620-Fill-Box. This performs sends the queue names to the
005280*      Queue List Box in the Print Dialog.
```

410 The COBOL Presentation Manager Programming Guide

```
005290*****
005300 620-Fill-Box.
005310      Move QueueNames(Indx) to Keyname
005320      Call OS2API 'WinSendDlgItemMsg' using
005330          by value     hwnd
005340          by value     ID-QueueListBox
005350          by value     LM-INSERTITEM
005360          by value     LIT-END
005370          by reference KeyName
005380          returning   ReturnData.
005390
005400*****
005410* 625-Fill-Box. This perform sends the driver names to the
005420*     Driver List Box in the Print Dialog.
005430*****
005440 625-Fill-Box.
005450      Move PrintDrivers(Indx) to Keyname
005460      Call OS2API 'WinSendDlgItemMsg' using
005470          by value     hwnd
005480          by value     ID-DriverListBox
005490          by value     LM-INSERTITEM
005500          by value     LIT-END
005510          by reference KeyName
005520          returning   ReturnData.
005530
005540*****
005550* 630-Extract-Drivers.
005560*
005570*****
005580 630-Extract-Drivers.
005590      Move UNSPtr to ShortWork
005600      Unstring PrtDrivers(Indx) Delimited by ValueComma
005610          or NullString
005620          into PrtString Delimiter in StopByte
005630          with Pointer UNSPtr
005640      End-Unstring
005650      Subtract ShortWork from UNSPtr giving ShortWork
005660      Move NullString to PrtStringWork(ShortWork)
005670      Move PrtString to PrintDrivers(DriverCount).
005680*****
```

```
*****  
/*  
/*      PRINTER CALL SPECIFICATION DIALOG HEADER FILE (PRINTER.H)  
/*  
/*  
/* *****  
/*  
/* This file defines the C symbolic constants used in the  
/* PRINTER Dialog file. The equivalent constants are defined  
/* to SAMPLE.CBL in the SAMPLE.CIN COBOL include file.  
/*  
*****  
  
#define ID_PrinterDlgBox          740  
#define ID_PrinterText1           741  
#define ID_QueueListBox           742  
#define ID_PrinterText2           743  
#define ID_DriverListBox          744  
#define ID_PrinterText3           745  
#define ID_PrinterQueue           746  
#define ID_QueueGpBox             747  
#define ID_PrinterText4           748  
#define ID_PrinterDriver          749  
#define ID_DriverGpBox            750  
#define ID_PrinterText5           751  
#define ID_PrinterCopies          752  
#define ID_PrinterButtonOK         753  
#define ID_PrinterButtonCan        754  
  
#define IDS_MsgBoxThree           803
```

```
DLGTEMPLATE ID_PrinterDlgBox
BEGIN
  DIALOG "Printer Setup", ID_PrinterDlgBox,
         9, 35, 294, 87,
         FS_NOBYTEALIGN | FS_DLGBORDER | FS_BORDER |
         WS_CLIPSIBLINGS | WS_SAVEBITS,
         FCF_TITLEBAR
  BEGIN
    CONTROL "Printer Queues", ID_PrinterText1,
            182, 72, 98, 7,
            WC_STATIC, SS_TEXT | DT_CENTER | DT_TOP |
            WS_GROUP | WS_VISIBLE
            WS_VISIBLE
    CONTROL "", ID_QueueListBox,
            180, 46, 105, 25,
            WC_LISTBOX, WS_TABSTOP | WS_VISIBLE
    CONTROL "Printer Drivers", ID_PrinterText2,
            181, 38, 99, 7,
            WC_STATIC, SS_TEXT | DT_CENTER | DT_TOP |
            WS_GROUP | WS_VISIBLE
            WS_VISIBLE
    CONTROL "", ID_DriverListBox,
            180, 12, 105, 25,
            WC_LISTBOX, WS_TABSTOP | WS_VISIBLE
    CONTROL "Printer Queue:", ID_PrinterText3,
            10, 72, 105, 7,
            WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
            WS_GROUP | WS_VISIBLE
    CONTROL "", ID_PrinterQueue,
            11, 61, 157, 8,
            WC_STATIC,
            SS_TEXT | DT_LEFT | DT_VCENTER |
            WS_GROUP | WS_VISIBLE
    CONTROL "", ID_QueueGpBox,
            8, 59, 162, 15,
            WC_STATIC,
            SS_GROUPBOX |
            WS_GROUP | WS_VISIBLE
    CONTROL "Printer Driver:", ID_PrinterText4,
            11, 51, 105, 7,
            WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
            WS_GROUP | WS_VISIBLE
    CONTROL "", ID_PrinterDriver,
            11, 40, 157, 8,
            WC_STATIC,
            SS_TEXT | DT_LEFT | DT_VCENTER |
            WS_GROUP | WS_VISIBLE
    CONTROL "", ID_DriverGpBox,
            8, 38, 162, 15,
            WC_STATIC,
            SS_GROUPBOX |
```

```
        WS_GROUP | WS_VISIBLE
CONTROL "Nbr of Copies", ID_PrinterText5,
        11, 30, 105, 7,
        WC_STATIC, SS_TEXT | DT_LEFT | DT_TOP |
        WS_GROUP | WS_VISIBLE
CONTROL "", ID_PrinterCopies,
        12, 17, 18, 8,
        WC_ENTRYFIELD, ES_RIGHT | ES_MARGIN |
        WS_TABSTOP | WS_VISIBLE
CONTROL "OK", ID_PrinterButtonOK,
        75, 14, 43, 13,
        WC_BUTTON, BS_PUSHBUTTON | WS_TABSTOP |
        WS_VISIBLE
CONTROL "Cancel", ID_PrinterButtonCan,
        124, 14, 43, 13,
        WC_BUTTON, BS_PUSHBUTTON | WS_TABSTOP |
        WS_VISIBLE
END
END
```


DosFreeModule

by value **hwndModule** [pic 9(4) comp-5] Module handle returned by DosLoadModule
 returning **ReturnCode** [pic 9(4) comp-5] Returned code

DosLoadModule

by reference **Buffer** [string pointer] Buffer that will receive the name of object that contributed to failure of the module load
 by value **BufferLength** [pic 9(4) comp-5] Length og the buffer in parameter one
 by reference **ModuleName** [string pointer] ASCII string name of the module to be loaded
 by reference **hwndModule** [pic 9(4) comp-5] Variable that will receive the module's handle
 returning **ReturnCode** [pic 9(4) comp-5] Return code

GpiCharStringAt

by value **hps** [pic s9(9) comp-5] The handle of the presentation space
 by reference **rect** [structure pointer] The x and y starting position structure
 by value **LongWork** [pic s9(9) comp-5] The count of string characters
 by reference **CharacterString** [string pointer] Pointer to the string to be drawn into the ps
 returning **Success** [pic s9(9) comp-5] Correlation/Error indicator
GPI-OK Successful
GPI-HITS Correlate hit(s)
GPI-ERROR Error

GpiCreateLogFont

by value **hps** [pic s9(9) comp-5] The handle of the presentation space
 by reference **Fatrs-Name** [string pointer] Pointer to the logical font name
 by value **LCID-Number** [pic s9(9) comp-5] Character-set local identifier
 by reference **Fatrs-Table** [structure pointer] Pointer to the Font Attribute Table
 returning **Match-indicator** [pic s9(9) comp-5] Match indicator
FONT-MATCH Font requirements successfully matched
FONT-DEFAULT Font requirements not matched, default font used
GPI-ERROR An error occurred

GpiDeleteSetId

by value **hps** [pic s9(9) comp-5] The handle of the presentation space
 by value **LCID-Number** [pic s9(9) comp-5] Character-set local identifier
 returning **Success** [pic 9(4) comp-5] Success indicator
True Successful completion
False Error occurred

GpiQueryFontMetrics

by value **hps** [pic s9(9) comp-5] The handle of the presentation space
 by value **MetricsLength** [pic s9(9) comp-5] The length of the metrics structure
 by reference **FontMetrics** [structure pointer] Pointer to the font metrics structure
 returning **Success** [pic 9(4) comp-5] Success indicator
True Successful completion
False Error occurred

GpiSetCharSet

by value **hps** [pic s9(9) comp-5] The handle of the presentation space
 by value **LCID-Number** [pic s9(9) comp-5] Character-set local identifier
 returning **Success** [pic 9(4) comp-5] Success indicator
True Successful completion
False Error occurred

GpiSetColor

by value **hps** [pic s9(9) comp-5] The handle of the presentation space
 by value **TextColor** [pic s9(9) comp-5] Color
 returning **Success** [pic 9(4) comp-5] Success Indicator
True Successful completion
False Error occurred

PrfCloseProfile

by value **Hini-Handle** [pic s9(9) comp-5] Profile handle
 returning **Success** [pic 9(4) comp-5] Success indicator
True Successful completion
False Error occurred

PrfOpenProfile

by value **hab** [pic s9(9) comp-5] Anchor-block handle
 by refreence **ProfileName** [string pointer] Pointer to the qualified profile name
 returning **Hini-Handle** [pic s9(9) comp-5] Profile handle
Null Error occurred
Other Profile file handle

PrfQueryProfileInt

by value **Hini-Handle** [pic s9(9) comp-5] Profile handle
 by reference **ApplName** [string pointer] Pointer to the application name string
 by reference **Key** [string pointer] Pointer to the key string
 by value **DefaultInteger** [pic s9(4) comp-5] The default integer value
 returning **Results** [pic s9(4) comp-5] Returned customization integer value

PrfQueryProfileString

by value Hini-Handle	[pic s9(9) comp-5]	Profile handle
by reference AppName	[string pointer]	Pointer to the application name string
by reference Key	[string pointer]	Pointer to the key string
by reference DefaultString	[string pointer]	Pointer to the default string
by reference ProfileString	[string pointer]	Pointer to the profile string buffer
by value MaxLength	[pic 9(9) comp-5]	The maximum length of the returned profile string
returning ReturnLength	[pic 9(9) comp-5]	The length of the returned profile string

PrfWriteProfileData

by value Hini-Handle	[pic s9(9) comp-5]	Profile handle
by reference AppName	[string pointer]	Pointer to the application name string
by reference Key	[string pointer]	Pointer to the key string
by reference BinaryCharacters	[string pointer]	Pointer to the binary data to be written to the profile
by value CharLength	[pic 9(9) comp-5]	Number of binary characters to write
returning Success	[pic 9(4) comp-5]	Success indicator
True	Successful completion	
False	Error occurred	

PrfWriteProfileString

by value Hini-Handle	[pic s9(9) comp-5]	Profile handle
by reference AppName	[string pointer]	Pointer to the application name string
by reference Key	[string pointer]	Pointer to the key string
by reference TextString	[string pointer]	Text string to be written to the profile
returning Success	[pic 9(4) comp-5]	Success indicator
True	Successful completion	
False	Error occurred	

WinBeginEnumWindows

by value hwndParentWindow	[pic s9(9) comp-5]	Parent window of children to be enumerated
returning hwndEnumeration	[pic s9(9) comp-5]	The window enumeration handle

WinBeginPaint

by value hwnd	[pic s9(9) comp-5]	The handle of the window to be painted
by value LongNull	[pic s9(9) comp-5]	The presentation space handle
Null	Obtain a cache presentation space	
Other	The presentation space handle	
by reference Rect	[structure pointer]	Pointer to the bounding rectangle structure
Null	No rectangle, repainting is not required	
Other	Specifies the smallest rectangle bounding the update region	
returning hps	[pic s9(9) comp-5]	Presentation space handle

WinCreateMsgQueue

by value **hab** [pic s9(9) comp-5] The Anchor-block handle
 by value **QueueSize** [pic s9(4) comp-5] The maximum number of queue entries
0 use the system default
Other Maximum number of queue entries
 returning **hmq** [pic s9(9) comp-5] The handle of the Message Queue
Null The queue can not be created
Other The message queue handle

WinCreateStdWindow

by value **hwnd Parent** [pic s9(9) comp-5] The handle of the parent-window
 by value **Style** [pic 9(9) comp-5] Combined window (WS-) and frame styles (FS-)
 by reference **FrameCreateFlags** [pic 9(9) comp-5] The Frame Create Flags
 by reference **ClientClassName** [pic x(x)] The client-window class name
 by reference **Title** [pic x(x)] The title-bar text
 by value **StyleClient** [pic 9(9) comp-5] The style of the client-window
 by value **Resources** [pic 9(4) comp-5] The resource identifier
Null The resource is bound into the .EXE
Other The resource module identifier
 by value **Id** [pic 9(4) comp-5] The individual resource identifier
 by reference **hwndClient** [pic s9(9) comp-5] The client-window handle
 returning **hwndFrame** [pic s9(9) comp-5] The frame-window handle

WinDefDigProc

by value **hwndDialog** [pic s9(9) comp-5] The dialog box handle
 by value **Msgid** [pic 9(4) comp-5] The message identity
 by value **MsgParm1** [pic 9(9) comp-5] Message parameter 1
 by value **MsgParm2** [pic 9(9) comp-5] Message parameter 2
 returning **Reply** [pic s9(9) comp-5] Message return data

WinDefWindowProc

by value **hwnd** [pic s9(9) comp-5] The window handle
 by value **Msgid** [pic 9(4) comp-5] The message identity
 by value **MsgParm1** [pic 9(9) comp-5] Message parameter 1
 by value **MsgParm2** [pic 9(9) comp-5] Message parameter 2
 returning **Reply** [pic s9(9) comp-5] Message return data

WinDestroyMsgQueue

by value **hmq** [pic s9(9) comp-5] The message queue handle
 returning **Destroyed** [pic 9(4) comp-5] The queue-destroyed indicator
True The message queue was destroyed
False The message queue was not destroyed

WinDestroyWindow

by value **hwndWindow** [pic s9(9) comp-5] The window handle
 returning **Success** [pic 9(4) comp-5] The window-destroyed indicator
True The window is destroyed
False The window was not destroyed

WinDismissDig

by value **hwndDialog** [pic s9(9) comp-5] The dialog box handle
 by value **RetrunData** [pic 9(4) comp-5] Returned to the caller of WinDlgBox or WinProcessDlg
True Dialog successfully dismissed
False Dialog not successfully dismissed

WinDispatchMsg

by value **hab** [pic s9(9) comp-5] The Anchor-block handle
 by reference **Qmsg** [structure pointer] A pointer to the message structure
 returning **Reply** [pic s9(9) comp-5] Data returned by the invoked window procedure

WinDlgBox

by value **hwndParent** [pic s9(9) comp-5] Parent-window handle
 by value **hwndOwner** [pic s9(9) comp-5] Owner-window handle
 by value **DialogProc** [procedure pointer] The dialog procedure entry point
 by value **Resourceld** [pic 9(4) comp-5] The resource identity
Null The resource is bound into the .EXE
Other The resource module identifier
 by value **DialogId** [pic 9(4) comp-5] Dialog-template identity
 by reference **CreateParms** [structure pointer] Application defined data area
 returning **DialogResult** [pic 9(4) comp-5] Dialog reply value

WinEndEnumWindows

by value **hwndEnumeration** [pic s9(9) comp-5] Enumeration handle
 returning **Success** [pic 9(4) comp-5] Success indicator
True Successful completion
False Error occurred

WinEndPaint

by value **hps** [pic s9(9) comp-5] The presentation space handle
 returning **Success** [pic 9(4) comp-5] Success indicator
True Successful completion
False An error occurred

WinFillRect

by value **hps** [pic s9(9) comp-5] The presentation space handle
 by reference **Rect** [structure pointer] The bounding structure of the rectangle to be filled
 by value **Color** [pic s9(9) comp-5] The color used to fill the rectangle
 returning **Success** [pic 9(4) comp-5] Success indicator
True Successful completion
False An error occurred

WinGetMsg

by value **hab** [pic s9(9) comp-5] The Anchor-block handle
 by reference **Qmsg** [structure pointer] A pointer to the message structure
 by value **FilterHandle** [pic s9(9) comp-5] The handle of the filter window
 by value **LowLimitMsg** [pic 9(9) comp-5] Lower limit for returned messages
 by value **HighLimitMsg** [pic 9(9) comp-5] Upper limit for returned messages
 returning **Result** [pic 9(4) comp-5] WM-Quit message indicator
True Message returned is not a WM-QUIT message
False Message returned is a WM-QUIT message

WinGetNextWindow

by value **hwndEnumeration** [pic s9(9) comp-5] The enumeration handle
 returning **hwndNextWindow** [pic s9(9) comp-5] Handle of the next window in the enumeration list
Null Error or all windows in the list have been enumerated
Other Next window handle

WinInitialize

by value **Options** [pic 9(4) comp-5] Not used - code as a null
 returning **hab** [pic s9(9) comp-5] Anchor-block handle
Null An error occurred
Other The Anchor-block handle

WinInvalidateRegion

by value **hwndWindow** [pic s9(9) comp-5] Handle of window with region to be changed
 by value **hwndRegion** [pic s9(9) comp-5] Handle of region to be added to update region
Null The whole window is to be added into the update region
Other Handle of the region to be added into the update region
 by value **ScopeIndicator** [pic 9(4) comp-5] Invalidation scope indicator
True Include the descendants of hwnd in the invalid rectangle
False Include descendants only if parent does not have WS_CLIPCHILDREN
 returning **Success** [pic 9(4) comp-5] Success indicator
True Successful completion
False Error occurred

WinLoadPointer

by value **hwndDesktop** [pic s9(9) comp-5] Desktop-window handle
 by value **Resource** [pic 9(4) comp-5] The resource identier
 by value **Pointer** [pic 9(4) comp-5] The pointer resource identifier
 returning **hwndPointer** [pic s9(9) comp-5] The pointer handle

WinLoadString

by value **hab** [pic s9(9) comp-5] The Anchor-block handle
 by value **Resource** [pic s9(4) comp-5] The resource identifier
 by value **String** [pic 9(4) comp-5] The resource string identifier
 by value **MaxBuffer** [pic s9(4) comp-5] The length of the buffer to hold the string
 by reference **Buffer** [buffer pointer] Pointer to the buffer to receive the resource string
 returning **StringLength** [pic s9(4) comp-5] The length of the returned string
0 An error occurred
Other The maximum value of (MaxBuffer -1)

WinMessageBox

by value **hwndParent** [pic s9(9) comp-5] Message box parent window handle
 by value **hwndOwner** [pic s9(9) comp-5] Owner window handle
 by reference **MessageBoxText** [string pointer] Pointer to the message box text
 by reference **MessageBoxTitle** [string pointer] Message box title text
 by value **MessageBoxIdentity** [pic 9(4) comp-5] Message box window identity
 by value **Style** [pic 9(4) comp-5] Message box window style
 returning **Response** [pic 9(4) comp-5] Response

WinPostMsg

by value **hwndWindow** [pic s9(9) comp-5] Target window handle
Null The message is posted to the queue associated with the current thread
Other Target window handle
 by value **MsgId** [pic 9(4) comp-5] Message identity
 by value **MsgParm1** [pic 9(9) comp-5] Message parameter 1
 by value **MsgParm2** [pic 9(9) comp-5] Message parameter 2
 returning **Result** [pic 9(4) comp-5] Message-posted indicator
True Message successfully posted
False Message could not be posted

WinQuerySysValue

by value **DesktopHandle** [pic s9(9) comp-5] Desktop-window handle
 by value **ValueId** [pic s9(4) comp-5] A system value identifier
 returning **SystemValue** [pic s9(9) comp-5] The returned system value
0 Error occurred
Other The requested system value

WinQueryFocus

by value **DesktopHandle** [pic s9(9) comp-5] Desktop-window handle
 by value **ShortNull** [pic 9(4) comp-5] Window lock state indicator
 returning **hwndFocusWindow** [pic s9(9) comp-5] Handle of the focus window

WinQueryWindowPos

by value **hwndWindow** [pic s9(9) comp-5] Target window handle
 by reference **SWP** [structure pointer] A pointer to the SWP structure
 returning **Success** [pic s9(4) comp-5] Success indicator
True Window position SWP structure returned
False An error occurred

WinQueryWindowRect

by value **hwndWindow** [pic s9(9) comp-5] Target window handle
 by reference **Rcl** [structure pointer] Pointer to the window rectangle structure
 returning **Rectangle-indicator** [pic 9(4) comp-5] Success indicator
True Rectangle successfully returned
False Rectangle not successfully returned

WinQueryWindowULong

by value **hwnd** [pic s9(9) comp-5] Handle of the window to be queried
 by value **IndexValue** [pic s9(4) comp-5] Zero-based index of window word to be queried
 returning **WindowWordData** [pic 9(9) comp-5] Value contained in window word

WinQueryWindowText

by value **hwndFrameWindow** [pic s9(9) comp-5] Handle of frame window containing the title-bar text
 by value **BufferLength** [pic s9(4) comp-5] Length of the receive buffer
 by reference **Buffer** [buffer pointer] Pointer to the buffer to receive the text string
 returning **TextLength** [pic s9(4) comp-5] Length of the returned text string

WinRegisterClass

by value **hab** [pic s9(9) comp-5] The Anchor-block handle
 by reference **ClassName** [pic x(x)] The window-class name
 by value **WindowProc** [procedure pointer] The window procedure entry point
 by value **ClassStyle** [pic 9(9) comp-5] Default window style for all windows of this class
 by value **WindowWords** [pic 9(4) comp-5] Extra application storage per window
 returning **Registered** [pic 9(4) comp-5] Window class registration indicator
True The window class was successfully registered
False The window class was not successfully registered

WinSendDlgItemMsg

by value **hwndDialog** [pic s9(9) comp-5] The target dialog window handle
 by value **Identity** [pic s9(4) comp-5] Identity of the control window
 by value **Msgid** [pic 9(4) comp-5] Message identity
 by value **MsgParm1** [pic 9(9) comp-5] Message parameter 1
 by value **MsgParm2** [pic 9(9) comp-5] Message parameter 2
 returning **RetrunData** [pic 9(4) comp-5] Message return data

WinSendMsg

by value **hwndWindow** [pic s9(9) comp-5] Target window handle
 by value **Msgid** [pic 9(4) comp-5] Message identity
 by value **MsgParam1** [pic 9(9) comp-5] Message Parameter 1
 by value **MsgParam2** [pic 9(9) comp-5] Message Parameter 2
 returning **MsgReturnData** [pic s9(4) comp-5] Message return data

WinSetDlgItemText

by value **hwndDialog** [pic s9(9) comp-5] The target dialog window handle
 by value **Identity** [pic s9(4) comp-5] Identity of the control window
 by reference **TextString** [pic x(x)] The dialog item text
 returning **Success** [pic 9(4) comp-5] Success indicator
True Dialog item text set
False An error occurred

WinSetPointer

by value **hwndDesktop** [pic s9(9) comp-5] Desktop-window handle
 by value **hwndPointer** [pic 9(9) comp-5] New pointer handle
 returning **Success** [pic s9(4) comp-5] Pointer-updated indicator
True Pointer successfully updated
False Pointer not successfully updated

WinSetPresParam

by value **hwndWindow** [pic s9(9) comp-5] The handle of the target window
 by value **PP-VALUE** [pic 9(9) comp-5] The presentation parameter to be altered
 by value **ByteCount** [pic 9(9) comp-5] Count of data passed in the AttrValue parameter
 by reference **ParamString** [string pointer] Pointer to the presentation parameter
 returning **Success** [pic 9(4) comp-5] Success indicator
True Successful completion
False Error occurred

WinSetWindowPos

by value **hwndWindow** [pic s9(9) comp-5] The window handle
 by value **Behind** [pic 9(9) comp-5] The relative window placement order
HWND-TOP Place window on top of sibling windows
HWND-BOTTOM Place window behind sibling windows
Other (handle) Place window behind this sibling window
 by value **X** [pic 9(4) comp-5] The window X-coordinate position
 by value **Y** [pic 9(4) comp-5] The window Y-coordinate position
 by value **CX** [pic 9(4) comp-5] The window size along the X-axis
 by value **CY** [pic 9(4) comp-5] The window size along the Y-axis
 by value **Options** [pic 9(4) comp-5] The window positioning options (SWP-)
 returning **Success** [pic 9(4) comp-5] Success indicator
True The window was successfully positioned
False The window was not successfully positioned

WinSetWindowText

by value **hwndFrame** [pic s9(9) comp-5] Handle of the frame window to receive the text
 by reference **TextString** [string pointer] Pointer to the text string
 returning **Success** [pic 9(4) comp-5] Success indicator
True Text updated
False Error Occurred

WinSetWindowULong

by value **hwnd** [pic s9(9) comp-5] Handle of window containing the window words
 by value **IndexValue** [pic s9(4) comp-5] Zero-based index value of window word
 by value **WindowWordData** [pic 9(9) comp-5] Value to be placed into the window word
 returning **Success** [pic 9(4) comp-5] Success indicator
True Successful completion
False Error indicator

WinSubclassWindow

by value **hwndOriginalProc** [pic s9(9) comp-5] Handle of window to be subclassed
 by value **NewProcAddr** [procedure pointer] The subclassing window procedure entry point
 returning **OriginalProcAddr** [procedure pointer] The subclassed window procedure entry point

WinTerminate

by value **hab** [pic s9(9) comp-5] The Anchor-block handle
 returning **Terminated** [pic 9(4) comp-5] Termination indicator
True The application's use of PM successfully terminated
False The application's use of PM not successfully terminated

WinUpper

by value **hab** [pic s9(9) comp-5] The Anchor-block handle
 by value **CodePage** [pic 9(4) comp-5] The code page number
Null Use the current process code page
Other Use the specified code page
 by value **CountryCode** [pic 9(4) comp-5] The country code number
Null Use the default country code in the CONFIG.SYS file
Other Use the specified country code
 by reference **TextString** [string pointer] Pointer to the string to be converted
 returning **ReturnLength** [pic 9(4) comp-5] The length of the converted string

WinWindowFromID

by value **hwndParent** [pic s9(9) comp-5] Parent-window handle
 by value **ChildWindIdentifier** [pic 9(4) comp-5] Identity of the child window
 returning **hwndWindow** [pic 9(9) comp-5] Requested window handle

PM Message	Value	PM Message	Value
WM_ACTIVATE	13	WM_EXTMOUSELAST	1048
WM_ADJUSTFRAMEPOS	86	WM_EXTMousereserved1	1043
WM_ADJUSTWINDOWPOS	8	WM_EXTMousereserved2	1046
WM_APPTERMINATENOTIFY	46	WM_EXTMousereserved3	1049
WM_BEGINDRAG	1056	WM_FLASHWINDOW	64
WM_BUTTONCLICKFIRST	113	WM_FOCUSCHANGE	67
WM_BUTTONCLICKLAST	121	WM_FORMATFRAME	65
WM_BUTTON1DBLCLK	115	WM_HELP	34
WM_BUTTON1DOWN	113	WM_HELPBASE *	3840
WM_BUTTON1MOTIONSTART	1041	WM_HELPTOP *	4095
WM_BUTTON1MOTIONEND	1042	WM_HITTEST	12
WM_BUTTON1UP	114	WM_HSCROLL	50
WM_BUTTON2DBLCLK	118	WM_HSCROLLCLIPBOARD	101
WM_BUTTON2DOWN	116	WM_INITDLG	59
WM_BUTTON2MOTIONSTART	1044	WM_INITMENU	51
WM_BUTTON2MOTIONEND	1045	WM_JOURNALNOTIFY	124
WM_BUTTON2UP	117	WM_MATCHMNEMONIC	61
WM_BUTTON3DBLCLK	121	WM_MEASUREITEM	55
WM_BUTTON3DOWN	119	WM_MENUEND	53
WM_BUTTON3MOTIONSTART	1047	WM_MENUSELECT	52
WM_BUTTON3MOTIONEND	1048	WM_MINMAXFRAME	70
WM_BUTTON3UP	120	WM_MOUSEFIRST	112
WM_CALCFRAMERECT	83	WM_MOUSELAST	121
WM_CALCVALIDRECTS	9	WM_MOUSEMOVE	112
WM_CHAR	122	WM_MOUSETRANSLATEFIRST	1056
WM_CHORD	1040	WM_MOUSETRANSLATELAST	1062
WM_CLOSE	41	WM_MOVE	6
WM_COMMAND	32	WM_NEXTMENU	78
WM_CONTEXTHELP	1061	WM_NULL	0
WM_CONTEXTMENU	1060	WM_OPEN	1059
WM_CONTROL	48	WM_OTHERWINDOWDESTROYED	3
WM_CONTROLPOINTER	56	WM_OWNERPOSCHANGE	82
WM_CREATE	1	WM_PAINT	35
WM_DESTROY	2	WM_PAINTCLIPBOARD	99
WM_DESTROYCLIPBOARD	98	WM_PRESPARAMCHANGED	47
WM_DRAGFIRST	784	WM_QUERYACCELTABLE	74
WM_DRAGLAST	815	WM_QUERYBORDERSIZE	77
WM_DRAWCLIPBOARD	103	WM_QUERYCONVERTPOS	176
WM_DRAWITEM	54	WM_QUERYDLGCODE	58
WM_ENABLE	4	WM_QUERYFOCUSCHAIN	81
WM_ENDDRAG	1057	WM_QUERYFRAMECTLCOUNT	89
WM_ERASEBACKGROUND	79	WM_QUERYFRAMEINFO	80
WM_ERROR	93	WM_QUERYHELPINFO	91
WM_EXTMOUSEFIRST	1040	WM_QUERYICON	72

* Messages 0x0F01 through 0x0FFE reserved for the Help Manager

PM Message	Value	PM Message	Value
WM_QUERYTRACKINFO	76		
WM_QUERYWINDOWPARAMS	11		
WM_QUIT	42		
WM_REALIZEPALETTE	94		
WM_RENDERALLFMTS	97		
WM_RENDERFMT	96		
WM_SAVEAPPLICATION	62		
WM_SELECT	1058		
WM_SEM1	37		
WM_SEM2	38		
WM_SEM3	39		
WM_SEM4	40		
WM_SETACCELTABLE	73		
WM_SETBORDERSIZE	68		
WM_SETFOCUS	15		
WM_SETHELPINFO	92		
WM_SETICON	71		
WM_SETSELECTION	16		
WM_SETWINDOWPARAMS	10		
WM_SHOW	5		
WM_SIZE	7		
WM_SIZECLIPBOARD	100		
WM_SUBSTITUTESTRING	60		
WM_SYSCOLORCHANGE	43		
WM_SYSCOMMAND	33		
WM_SYSVALUECHANGED	45		
WM_TEXTEDIT	1062		
WM_TIMER	36		
WM_TRACKFRAME	69		
WM_TRANSLATEACCEL	75		
WM_UPDATEFRAME	66		
WM_USER	4096		
WM_VIOCHAR	123		
WM_VSCROLL	49		
WM_VSCROLLCLIPBOARD	102		
WM_WINDOWPOSCHANGED	85		

PM Header Definition	Value	PM Header Definition	Value
Pattern Bundle Masks		AWP_MINIMIZED	0x00010000L
ABB_BACK_COLOR	0x0002L	AWP_RESTORED	0x00040000L
ABB_BACK_MIX_MODE	0x0008L	GpiBeginArea Control Flags	
ABB_COLOR	0x0001L	BA_ALTERNATE	0L
ABB_MIX_MODE	0x0004L	BA_BOUNDARY	0x0001L
ABB_REF_POINT	0x0040L	BA_NOBOUNDARY	0L
ABB_SET	0x0010L	BA_WINDING	0x0002L
ABB_SYMBOL	0x0020L	GpiBitBlt Bit Options	
Accelerator Key Flag		BBO_AND	1L
AF_ALT	0x0020	BBO_IGNORE	2L
AF_CHAR	0x0001	BBO_OR	0L
AF_CONTROL	0x0010	BITMAPINFO Structure Values	
AF_HELP	0x0200	BCA_HUFFMAN1D	3L
AF_LONEKEY	0x0040	BCA_RLE4	2L
AF_SCANCODE	0x0004	BCA_RLE8	1L
AF_SHIFT	0x0008	BCA_RLE24	4L
AF_SYSCOMMAND	0x0100	BCA_UNCOMP	0L
AF_VIRTUALKEY	0x0002	BCE_PALETTE	(-1L)
GpiSetAttrMode Modes		BCE_RGB	0L
AM_ERROR	(-1L)	BRA_BOTTOMUP	0L
AM_PRESERVE	0L	BRH_ERRORDIFFUSION	1L
AM_NOPRESERVE	1L	BRH_NOHALFTONED	0L
Help Manager Bitmap Alignment Flags		BRH_PANDA	2L
ART_CENTER	0x04L	BRH_SUPERCIRCLE	3L
ART_LEFT	0x01L	BRU_METRIC	0L
ART_RIGHT	0x02L	Button Drawing State Codes	
ART_RUNIN	0x10L	BDS_DEFAULT	0x0400
Segment Attributes		BDS_DISABLED	0x0200
ATTR_CHAINED	6L	BDS_HIITED	0x0100
ATTR_DETECTABLE	1L	BITMAPFILEHEADER Type Values	
ATTR_DYNAMIC	8L	BFT_ICON	0x4349
ATTR_ERROR	(-1L)	BFT_BMAP	0x4d42
ATTR_FASTCHAIN	9L	BFT_POINTER	0x5450
ATTR_OFF	0L	BFT_COLORICON	0x4943
ATTR_ON	1L	BFT_COLORPOINTER	0x5043
ATTR_PROP_DETECTABLE	10L	BFT_BITMAPARRAY	0x4142
ATTR_PROP_VISIBLE	11L	Notebook Window Attributes	
ATTR_VISIBLE	2L	BKA_ALL	0x0001
Window Position Return Codes		BKA_AUTOPAGESIZE	0x0100
AWP_ACTIVATE	0x00080000L	BKA_BITMAP	0x0800
AWP_DEACTIVATE	0x00100000L	BKA_END	0x0200
AWP_MAXIMIZED	0x00020000L		

PM Header Definition	Value	PM Header Definition	Value
BKA_FIRST	0x0004	BKN_NEWPAGESIZE	131
BKA_LAST	0x0002	BKN_PAGEDELETED	133
BKA_MAJOR	0x0040	BKN_PAGELSELECTED	130
BKA_MAJORTAB	0x0001		Notebook Window Styles
BKA_MINOR	0x0080		
BKA_MINORTAB	0x0002	BKS_BACKPAGESBL	0x00000002
BKA_NEXT	0x0008	BKS_BACKPAGESBR	0x00000001
BKA_PAGEBUTTON	0x0100	BKS_BACKPAGESTL	0x00000008
BKA_PREV	0x0010	BKS_BACKPAGESTR	0x00000004
BKA_SINGLE	0x0002	BKS_MAJORTABBOTTOM	0x00000080
BKA_STATUSTEXTON	0x0001	BKS_MAJORTABLEFT	0x00000020
BKA_TAB	0x0004	BKS_MAJORTABRIGHT	0x00000010
BKA_TEXT	0x0400	BKS_MAJORTABTOP	0x00000040
BKA_TOP	0x0020	BKS_POLYGON TABS	0x000000400
		BKS_ROUNDED TABS	0x000000200
		BKS_STATUSTEXTCENTER	0x00004000
Notebook Color Presentation Parameters		BKS_STATUSTEXTLEFT	0x00001000
BKA_BACKGROUND PAGE COLOR INDEX	0x0001	BKS_STATUSTEXTRIGHT	0x00002000
BKA_BACKGROUND PAGE COLOR	0x0002	BKS_SQUARE TABS	0x000000100
BKA_BACKGROUND MAJOR COLOR INDEX	0x0003	BKS_TABTEXTCENTER	0x00040000
BKA_BACKGROUND MAJOR COLOR	0x0004	BKS_TABTEXTLEFT	0x00010000
BKA_BACKGROUND MINOR COLOR INDEX	0x0005	BKS_TABTEXTRIGHT	0x00020000
BKA_BACKGROUND MINOR COLOR	0x0006		Button Messages
BKA_FOREGROUND MAJOR COLOR INDEX	0x0007	BM_CLICK	0x0120
BKA_FOREGROUND MAJOR COLOR	0x0008	BM_QUERYCHECK	0x0124
BKA_FOREGROUND MINOR COLOR INDEX	0x0009	BM_QUERYCHECKINDEX	0x0121
BKA_FOREGROUND MINOR COLOR	0x000A	BM_QUERYHILITE	0x0122
		BM_SETCHECK	0x0125
Nootbook Messages		BM_SETDEFAULT	0x0126
BKM_CALCPAGERECT	0x0356	BM_SETHILITE	0x0123
BKM_DELETEPAGE	0x0357		Background Mixes
BKM_INSERTPAGE	0x0358	BM_AND	6L
BKM_INVALIDATETABS	0x0359	BM_DEFAULT	0L
BKM_QUERYPAGECOUNT	0x035b	BM_ERROR	(-1L)
BKM_QUERYPAGE DATA	0x035d	BM_INVERT	12L
BKM_QUERYPAGEID	0x035c	BM_LEAVEALONE	5L
BKM_QUERYPAGESTYLE	0x0368	BM_MASKSRCNOT	8L
BKM_QUERYPAGEWINDOWHWND	0x035e	BM_MERGENOTSRC	15L
BKM_QUERYTABBITMAP	0x035f	BM_MERGESRCNOT	13L
BKM_QUERYTABTEXT	0x0360	BM_NOTCOPYSRC	14L
BKM_SETDIMENSIONS	0x0361	BM_NOTMASKSRC	16L
BKM_SETNOTEBOOKCOLORS	0x0367	BM_NOTMERGESRC	10L
BKM_SETPAGEDATA	0x0362	BM_NOTXORSRC	11L
BKM_SETPAGEWINDOWHWND	0x0363	BM_ONE	17L
BKM_SETSTATUSLINE TEXT	0x0364	BM_OR	1L
BKM_SETTABBITMAP	0x0365	BM_OVERPAINT	2L
BKM_SETTABTEXT	0x0366	BM_SUBTRACT	7L
BKM_TURNTOPAGE	0x035a	BM_XOR	4L
Notebook Notification Messages			
BKN_HELP	132		

PM Header Definition	Value	PM Header Definition	Value
BM_ZERO	9L	DevQueryCaps Device Codes	
	Broadcast Message Codes	CAPS_ADDITIONAL_GRAPHICS	32L
BMSG_DESCENDANTS	0x0004	CAPS_BACKGROUND_MIX_SUPPORT	20L
BMSG_FRAMEONLY	0x0008	CAPS_BITMAP_FORMATS	23L
BMSG_POST	0x0000	CAPS_BM_AND	32L
BMSG_POSTQUEUE	0x0002	CAPS_BM_GENERAL_BOOLEAN	64L
BMSG_SEND	0x0001	CAPS_BM_LEAVEALONE	16L
	Button Notification Messages	CAPS_BM_OR	1L
BN_CLICKED	1	CAPS_BM_OVERPAINT	2L
BN_DBLCLICKED	2	CAPS_BM_XOR	8L
BN_PAINT	3	CAPS_BYTE_ALIGN_NOT_REQUIRED	2L
	Button Styles	CAPS_BYTE_ALIGN_RECOMMENDED	1L
BS_AUTOCHECKBOX	2L	CAPS_BYTE_ALIGN_REQUIRED	0L
BS_AUTORADIOBUTTON	4L	CAPS_CHAR_HEIGHT	11L
BS_AUTOSIZE	0x4000L	CAPS_CHAR_WIDTH	10L
BS_AUTO3STATE	6L	CAPS_COLOR_INDEX	34L
BS_BITMAP	0x0040L	CAPS_COLTABL_REALIZE	8L
BS_CHECKBOX	1L	CAPS_COLTABL_RGB_8	1L
BS_DEFAULT	0x0400L	CAPS_COLTABL_RGB_8_PLUS	2L
BS_HELP	0x0100L	CAPS_COLTABL_TRUE_MIX	4L
BS_ICON	0x0080L	CAPS_COLOR_BITCOUNT	16L
BS_NOBORDER	0x1000L	CAPS_COLOR_CURSOR_SUPPORT	128L
BS_NOCURSORSELECT	0x2000L	CAPS_COLOR_PLANES	15L
BS_NOPOINTERFOCUS	0x0800L	CAPS_COLOR_TABLE_SUPPORT	17L
BS_PRIMARYSTYLES	0x000fL	CAPS_COLORS	14L
BS_PUSHBUTTON	0L	CAPS_COSMETIC_WIDELINE_SUPPORT	512L
BS_RADIobutton	3L	CAPS_DEV_WINDOWING_SUPPORT	1L
BS_SYSCOMMAND	0x0200L	CAPS_DEVICE_FONT_SIM	39L
BS_USERBUTTON	7L	CAPS_DEVICE_FONTS	27L
BS_3STATE	5L	CAPS_DEVICE_WINDOWING	31L
	Container Control Attributes	CAPS_DRIVER_VERSION	3L
CA_CONTAINERTITLE	0x00020000L	CAPS_FAMILY	0L
CA_DETAILSVIEWTITLES	0x00800000L	CAPS_FM_AND	32L
CA_DRAWBITMAP	0x02000000L	CAPS_FM_GENERAL_BOOLEAN	64L
CA_DRAWICON	0x04000000L	CAPS_FM_LEAVEALONE	16L
CA_MIXEDTARGETEMPH	0x20000000L	CAPS_FM_OR	1L
CA_ORDEREDTARGETEMPH	0x01000000L	CAPS_FM_OVERPAINT	2L
CA_OWNERDRAW	0x00400000L	CAPS_FM_XOR	8L
CA_OWNERPAINTBACKGROUND	0x10000000L	CAPS_FONT_IMAGE_DEFAULT	8L
CA_TITLECENTER	0x00200000L	CAPS_FONT_OUTLINE_DEFAULT	4L
CA_TITLELEFT	0x00080000L	CAPS_FOREGROUND_MIX_SUPPORT	19L
CA_TITLE_READONLY	0x08000000L	CAPS_GRAPHICS_CHAR_HEIGHT	36L
CA_TITLERIGHT	0x00100000L	CAPS_GRAPHICS_CHAR_WIDTH	35L
CA_TITLESEPARATOR	0x00040000L	CAPS_GRAPHICS_KERNING_SUPPORT	2L

PM Header Definition	Value	PM Header Definition	Value
CAPS_IO_DUMMY	1L	Compression/Decompression Command Constants	
CAPS_IO_SUPPORTS_IP	3L	CBD_BITS	0L
CAPS_IO_SUPPORTS_IO	4L	CBD_COLOR_CONVERSION	0x00000001L
CAPS_IO_SUPPORTS_OP	2L	CBD_COMPRESSION	1L
CAPS_LINEWIDTH_THICK	40L	CBD_DECOMPRESSION	2L
CAPS_MARKER_HEIGHT	25L	Combo Box Messages	
CAPS_MARKER_WIDTH	26L	CBM_HILITE	0x0171L
CAPS_MOUSE_BUTTONS	18L	CBM_ISLISTSHOWING	0x0172L
CAPS_PALETTE_MANAGER	256L	CBM_SHOWLIST	0x0170L
CAPS_PHYS_COLORS	33L	Combo Box Notification Messages	
CAPS_RASTER_BANDING	2L	CBN_EFCHANGE	1
CAPS_RASTER_BITBLT	1L	CBN_EFSCROLL	2
CAPS_RASTER_BITBLT_SCALING	4L	CBN_ENTER	7
CAPS_RASTER_CAPS	24L	CBN_LBSELECT	4
CAPS_RASTER_FLOOD_FILL	64L	CBN_LBSCROLL	5
CAPS_RASTER_FONTS	32L	CBN_MEMERROR	3
CAPS_RASTER_SET_PEL	16L	CBN_SHOWLIST	6
CAPS_SCALED_DEFAULT_MARKERS	64L	Combo Box Styles	
CAPS_SMALL_CHAR_HEIGHT	13L	CBS_DROPDOWN	0x0002L
CAPS_SMALL_CHAR_WIDTH	12L	CBS_DROPDOWNLIST	0x0004L
CAPS_TECHNOLOGY	2L	CBS_SIMPLE	0x0001L
CAPS_TECH_POSTSCRIPT	5L	Clipboard Formats	
CAPS_TECH_RASTER_CAMERA	4L	CF_BITMAP	2
CAPS_TECH_RASTER_DISPLAY	2L	CF_DSPBITMAP	4
CAPS_TECH_RASTER_PRINTER	3L	CF_DSPMETAFILE	6
CAPS_TECH_UNKNOWN	0L	CF_DSPTEXT	3
CAPS_TECH_VECTOR_PLOTTER	1L	CF_METAFILE	5
CAPS_VDD_DDB_TRANSFER	1L	CF_PALETTE	9
CAPS_VERTICAL_FONT_RES	38L	CF_TEXT	1
CAPS_VERTICAL_RESOLUTION	9L	Container Control Attributes For Data And Titles	
CAPS_VIO_LOADABLE_FONTS	21L	CFA_BITMAPORICON	0x00000100L
CAPS_WIDTH	4L	CFA_BOTTOM	0x00000020L
CAPS_WIDTH_IN_CHARS	6L	CFA_CENTER	0x00000004L
CAPS_WINDOW_BYTE_ALIGNMENT	22L	CFA_DATE	0x00002000L
Character Bundle Mask Bits		CFA_FIREADONLY	0x00008000L
CBB_ANGLE	0x0080L	CFA_FITITLEREADONLY	0x00010000L
CBB_BACK_COLOR	0x0002L	CFA_HORZSEPARATOR	0x00000040L
CBB_BACK_MIX_MODE	0x0008L	CFA_INVISIBLE	0x000000040L
CBB_BOX	0x0040L	CFA_LEFT	0x00000001L
CBB_BREAK_EXTRA	0x1000L	CFA_OWNER	0x00001000L
CBB_COLOR	0x0001L	CFA_RIGHT	0x00000002L
CBB_DIRECTION	0x0200L	CFA_SEPARATOR	0x00000200L
CBB_EXTRA	0x0800L		
CBB_MIX_MODE	0x0004L		
CBB_MODE	0x0020L		
CBB_RESERVED	0x0400L		
CBB_SET	0x0010L		
CBB_SHEAR	0x0100L		

PM Header Definition	Value	PM Header Definition	Value
CFA_STRING	0x000000800L	CLR_DARKGREEN	12L
CFA_TIME	0x00004000L	CLR_DARKPINK	11L
CFA_TOP	0x00000008L	CLR_DARKRED	10L
CFA_VCENTER	0x00000010L	CLR_DEFAULT	(-3L)
Clip Board Data Flags		CLR_ERROR	(-255L)
CFI_HANDLE	0x0200	CLR_FALSE	(-5L)
CFI_OWNERDISPLAY	0x0002	CLR_GREEN	4L
CFI_OWNERFREE	0x0001	CLR_NEUTRAL	7L
CFI_POINTER	0x0400	CLR_PALEGREY	15L
Character Directions		CLR_PINK	3L
CHDIRN_BOTTOMTOP	4L	CLR_RED	2L
CHDIRN_DEFAULT	0L	CLR_TRUE	(-4L)
CHDIRN_ERROR	(-1L)	CLR_UNCHANGED	(-6L)
CHDIRN_LEFTRIGHT	1L	CLR_WHITE	(-2L)
CHDIRN_RIGHTLEFT	3L	CLR_YELLOW	6L
CHDIRN_TOPBOTTOM	2L	Dialog Color Code definitions	
GpiCharStringPosAt Format Options		CLRC_BACKGROUND	2L
CHS_CLIP	0x0010L	CLRC_FOREGROUND	1L
CHS_LEAVEPOS	0x0008L	Character Modes	
CHS_OPAQUE	0x0001L	CM_DEFAULT	0L
CHS_STRIKEOUT	0x0400L	CM_ERROR	(-1L)
CHS_UNDERSCORE	0x0200L	CM_MODE1	1L
CHS_VECTOR	0x0002L	CM_MODE2	2L
Container Control Child Window IDs		CM_MODE3	3L
CID_BLANKBOX	0x7FF2	Container Control Messages	
CID_CNRTITLEWND	0x7FF5	CM_ALLOCDATAILFIELDINFO	0x0330
CID_HSCROLL	0x7FF3	CM_ALLOCRECORD	0x0331
CID_LEFTDVIEWND	0x7FF7	CM_ARRANGE	0x0332
CID_LEFTCOLTITLEWND	0x7FF6	CM_CLOSEEDIT	0x0381
CID_MLE	0x7FFA	CM_ERASERECORD	0x0333
CID_RIGHTCOLTITLEWND	0x7FF1	CM_FILTER	0x0334
CID_RIGHTDVIEWND	0x7FF8	CM_FREEDETAILFIELDINFO	0x0335
CID_RIGHTHSCROLL	0x7FF4	CM_FREERECORD	0x0336
CID_VSCROLL	0x7FF9	CM_HORZSCROLLSLPLITWINDOW	0x0337
Default Color Table Indices		CM_INSERTDETAILFIELDINFO	0x0338
CLR_BACKGROUND	0L	CM_INSERTRECORD	0x0339
CLR_BLACK	(-1L)	CM_INVALIDATEDETAILFIELDINFO	0x033a
CLR_BLUE	1L	CM_INVALIDATERECORD	0x033b
CLR_BROWN	14L	CM_OPENDIT	0x0380
CLR_CYAN	5L	CM_PAINTBACKGROUND	0x033c
CLR_DARKBLUE	9L	CM_QUERYCNRINFO	0x033d
CLR_DARKCYAN	13L	CM_QUERYDETAILFIELDINFO	0x033e
CLR_DARKGRAY	8L	CM_QUERYDRAGIMAGE	0x033f

PM Header Definition	Value	PM Header Definition	Value
CM_QUERYVIEWPORTRECT	0x0344	WM_COMMAND Message Source Codes	
CM_REMOVEDETAILFIELDINFO	0x0345	CMDSRC_ACCELERATOR	3
CM_REMOVERECORD	0x0346	CMDSRC_COLORDLG	7
CM_SCROLLWINDOW	0x0347	CMDSRC_FILEDLG	5
CM_SEARCHSTRING	0x0348	CMDSRC_FONTDLG	4
CM_SETCNRINFO	0x0349	CMDSRC_MENU	2
CM_SETRECORDEMPHASIS	0x034a	CMDSRC_OTHER	0
CM_SORTRECORD	0x034b	CMDSRC_PRINTDLG	6
Container Control Attributes		CMDSRC_PUSHBUTTON	1
CMA_BOTTOM	0x0001	Container Control Notification Messages	
CMA_CNRTITLE	0x0001	CN_BEGINEDIT	114
CMA_COMPLETE	0x0002	CN_CONTEXTMENU	117
CMA_DELTA	0x0002	CN_DRAGAFTER	101
CMA_DELTABOT	0x0002	CN_DRAGLEAVE	102
CMA_DELTAEND	0x0008	CN_DRAGOVER	103
CMA_DELTAHOME	0x0004	CN_DROP	104
CMA_DELTATOP	0x0001	CN_DROPHLP	105
CMA_END	0x0004	CN_EMPHASIS	108
CMA_ERASE	0x0008	CN_ENDEDIT	115
CMA_FIRST	0x0008	CN_ENTER	106
CMA_FIRSTINVIEW	0x0004	CN_INITDRAG	107
CMA_FLWINDOWATTR	0x0008	CN_KILLFOCUS	109
CMA_FONTMETRICS	0x0010	CN_QUERYDELT A	111
CMA_FREE	0x0001	CN_REALLOCPSZ	113
CMA_HORIZONTAL	0x0010	CN_SCROLL	110
CMA_ICON	0x0020	CN_SETFOCUS	112
CMA_INVALIDATE	0x0002	CN_HELP	116
CMA_ITEMORDER	0x0001	Container Control Record Attributes	
CMA_LAST	0x0040	CRA_CURSORRED	0x00000004L
CMA_LEFT	0x0080	CRA_DROPOENABLE	0x00000020L
CMA_LINESPACING	0x0020	CRA_FILTERED	0x00000010L
CMA_NEXT	0x0100	CRA_INUSE	0x00000008L
CMA_NOREPOSITION	0x0001	CRA_RECORDREADONLY	0x00000040L
CMA_PARTIAL	0x0200	CRA_SELECTED	0x00000001L
CMA_PFIELDINLAST	0x0040	CRA_TARGET	0x00000002L
CMA_PFIELDINFOBJECT	0x1000	GpiPlayMetaFile Options	
CMA_PREV	0x0400	CREA_DEFAULT	0L
CMA_PSORTRECORD	0x0080	CREA_DOREALIZE	3L
CMA_PTLOIGIN	0x0100	CREA_NOREALIZE	2L
CMA_REPOSITION	0x0002	CREA_REALIZE	1L
CMA_RIGHT	0x0800	CTAB_DEFAULT	0L
CMA_SLBITMAPORICON	0x0200	CTAB_NOMODIFY	1L
CMA_TEXT	0x1000	CTAB_REPLACE	3L
CMA_TEXTCHANGED	0x0004	CTAB_REPLACEPALETTE	4L
CMA_TOP	0x2000	DDEF_DEFAULT	0L
CMA_VERTICAL	0x4000	DDEF_IGNORE	1L
CMA_WINDOW	0x0002		
CMA_WORKSPACE	0x0004		
CMA_XVERTSPLITBAR	0x0400		
CMA_ZORDER	0x0008		

PM Header Definition	Value	PM Header Definition	Value
DDEF_LOADDISC	3L		
DOBJ_DEFAULT	0L		
DOBJ_DELETE	2L		
DOBJ_NODELETE	1L		
LC_DEFAULT	0L		
LC_LOADDISC	3L		
LC_NOLOAD	1L		
LT_DEFAULT	0L		
LT_NOMODIFY	1L		
LT_ORIGINALVIEW	4L		
RES_DEFAULT	0L		
RES_NORESET	1L		
RES_RESET	2L		
RS_DEFAULT	0L		
RS_NODISCARD	1L		
RSP_DEFAULT	0L		
RSP_NODISCARD	1L		
SUP_DEFAULT	0L		
SUP_NOSUPPRESS	1L		
SUP_SUPPRESS	2L		
GpiCombineRegion Options			
CRGN_AND	6L		
CRGN_COPY	2L		
CRGN_DIFF	7L		
CRGN_OR	1L		
CRGN_XOR	4L		
Container Control Styles			
CS_AUTOPOSITION	0x00000800L		
CS_EXTENDSEL	0x00000100L		
CS_MULTIPLESEL	0x0000200L		
CS_READONLY	0x00004000L		
CS_SINGLESEL	0x00000400L		
CS_VERIFYPOINTERS	0x00002000L		
Class Styles			
CS_CLIPCHILDREN	0x20000000L		
CS_CLIPSIBLINGS	0x10000000L		
CS_FRAME	0x00000020L		
CS_HITTEST	0x00000008L		
CS_MOVENOTIFY	0x00000001L		
CS_PARENTCLIP	0x08000000L		
CS_PUBLIC	0x00000010L		
CS_SAVEBITS	0x04000000L		
CS_SIZEREDRAW	0x00000004L		
CS_SYNCPAINT	0x02000000L		
Cursor Create Flags			
CURSOR_FLASH	0x0004		
CURSOR_FRAME	0x0002		
CURSOR_HALFTONE	0x0001		
CURSOR_SETPOS	0x8000		
CURSOR_SOLID	0x0000		
Container Control View Identifiers			
CV_DETAIL	0x00000008L		
CV_FLOW	0x00000010L		
CV_ICON	0x00000004L		
CV_MINI	0x00000020L		
CV_NAME	0x00000002L		
CV_TEXT	0x00000001L		
Window Bit Alignment Controls			
CVR_ALIGNBOTTOM	0x0002		
CVR_ALIGNLEFT	0x0001		
CVR_ALIGNRIGHT	0x0004		
CVR_ALIGNTOP	0x0008		
CVR_REDRAW	0x0010		
GpiConvert PS Coordinates			
CVTC_DEFAULTPAGE	3L		
CVTC_DEVICE	5L		
CVTC_MODEL	2L		
CVTC_PAGE	4L		
CVTC_WORLD	1L		
Border Drawing Flags			
DB_AREAATTRS	0x0010		
DB_AREAMIXMODE	0x0003		
DB_DESTINVERT	0x0002		
DB_DLGBORDER	0x0200		
DB_INTERIOR	0x0008		
DB_PATCOPY	0x0000		
DB_PATINVERT	0x0001		
DB_ROP	0x0007		
DB_STANDARD	0x0100		
Bitmap Drawing Flags			
DBM_HALFTONE	0x0002		
DBM_IMAGEATTRS	0x0008		
DBM_INVERT	0x0001		
DBM_NORMAL	0x0000		
DBM_STRETCH	0x0004		

PM Header Definition	Value	PM Header Definition	Value
Drag Source Object Control Flags		DEVESC_STD_JOURNAL	32600L
DC_CONTAINER	0x0008	Drag File Complete Flags	
DC_GROUP	0x0004	DF_MOVE	0x0001
DC_OPEN	0x0001	DF_SOURCE	0x0002
DC_PREPARE	0x0010	DF_SUCCESSFUL	0x0004
DC_REF	0x0002	Drag Error Option IDs	
DC_REMOVEABLEMEDIA	0x0020	DFF_COPY	2
GpiSetDrawControl Control Parameters		DFF_DELETE	3
DCTL_BOUNDARY	3L	DFF_MOVE	1
DCTL_CORRELATE	5L	GpiPutData/GetData Data Type	
DCTL_DISPLAY	2L	DFORM_PCLONG	4L
DCTL_DYNAMIC	4L	DFORM_PCSHORT	2L
DCTL_ERASE	1L	DFORM_NOCONV	0L
DCTL_ERROR	-1L	DFORM_S370SHORT	1L
DCTL_OFF	0L	Dialog Box Return Codes	
DCTL_ON	1L	DID_APPLY_BUTTON	311
Dynamic Data Exchange Status Field Constants		DID_CANCEL	2
DDE_FACK	0x0001	DID_CANCEL_BUTTON	2
DDE_FACKREQ	0x0008	DID_CANCEL_PB	2
DDE_FAPPSTATUS	0xFF00	DID_DIRECTORY_LB	264
DDE_FBUSY	0x0002	DID_DIRECTORY_TXT	263
DDE_FNODATA	0x0004	DID_DISPLAY_FILTER	303
DDE_FRESERVED	0x00C0	DID_DRIVE_CB	260
DDE_FRESPONSE	0x0010	DID_DRIVE_TXT	259
DDE_NOTPROCESSED	0x0020	DID_EMPHASIS_GROUPBOX	317
DDEFMT_TEXT	0x0001	DID_ERROR	0xffff
DevEscape Codes		DID_FILE_DIALOG	256
DEVESC_ABORTDOC	8153L	DID_FILENAME_ED	258
DEVESC_BREAK_EXTRA	16999L	DID_FILENAME_TXT	257
DEVESC_CHAR_EXTRA	16998L	DID_FILES_LB	266
DEVESC_DBE_FIRST	24450L	DID_FILES_TXT	265
DEVESC_DBE_LAST	24455L	DID_FILTER_CB	262
DEVESC_DRAFTMODE	16301L	DID_FILTER_TXT	261
DEVESC_ENDDOC	8151L	DID_FONT_DIALOG	300
DEVESC_ERROR	(-1L)	DID_HELP_BUTTON	310
DEVESC_FLUSHOUTPUT	16302L	DID_HELP_PB	267
DEVESC_GETCP	8000L	DID_NAME	301
DEVESC_GETSCALINGFACTOR	1L	DID_NAME_PREFIX	313
DEVESC_NEWFRAME	16300L	DID_OK	1
DEVESC_NEXTBAND	8152L	DID_OK_BUTTON	1
DEVESC_NOTIMPLEMENTED	0L	DID_OK_PB	1
DEVESC_QUERYESCSUPPORT	0L	DID_OUTLINE	307
DEVESC_QUERYVIOCELLSIZES	2L	DID_PRINTER_FILTER	304
DEVESC_RAWDATA	16303L	DID_RESET_BUTTON	312
DEVESC_SETMODE	16304L	DID_SAMPLE	306
DEVESC_STARTDOC	8150L		

PM Header Definition	Value	PM Header Definition	Value
DID_SAMPLE_GROUPBOX	316	Drag Transfer Response Flags	
DID_SIZE	305	DMFL_NATIVERENDER	0x0004
DID_SIZE_PREFIX	315	DMFL_RENDERFAIL	0x0020
DID_STRIKEOUT	309	DMFL_RENDEROK	0x0010
DID_STYLE	302	DMFL_RENDERRETRY	0x0008
DID_STYLE_PREFIX	314	DMFL_TARGETFAIL	0x0002
DID_UNDERSCORE	308	DMFL_TARGETSUCCESSFUL	0x0001
Dialog Code Information Flags		Drag Supported Operation Flags	
DLGC_BUTTON	0x0002	DO_COPY	0x0010
DLGC_CHECKBOX	0x0040	DO_COPYABLE	0x0001
DLGC_DEFAULT	0x0010	DO_DEFAULT	0xBFFE
DLGC_ENTRYFIELD	0x0001	DO_LINK	0x0018
DLGC_MENU	0x0100	DO_LINKABLE	0x0004
DLGC_MLE	0x0400	DO_MOVE	0x0020
DLGC_PUSHBUTTON	0x0020	DO_MOVEABLE	0x0002
DLGC_RADIOBUTTON	0x0004	DO_UNKNOWN	0xBFFF
DLGC_SCROLLBAR	0x0080	Dragover Response Codes	
DLGC_STATIC	0x0008	DOR_DROP	0x0001
DLGC_TABONCLICK	0x0200	DOR_NEVERDROP	0x0003
Drag And Drop Messages		DOR_NODROP	0x0000
DM_DRAGERROR	0x0324	DOR_NODROPOP	0x0002
DM_DRAGFILECOMPLETE	0x0326	Pointer Draw Constants	
DM_DRAGLEAVE	0x032d	DP_HALFTONED	0x0001
DM_DRAGOVER	0x032e	DP_INVERTED	0x0002
DM_DROP	0x032f	DP_NORMAL	0x0000
DM_DROPHHELP	0x032c	DevPostDeviceModes Codes	
DM_EMPHASIZETARGET	0x0325	DPDM_CHANGEPROP	1L
DM_ENDCONVERSATION	0x032b	DPDM_ERROR	(-1L)
DM_FILERENDERED	0x0323	DPDM_NONE	0L
DM_PRINT	0x032a	DPDM_POSTJOBPROP	0L
DM_RENDER	0x0329	DPDM_QUERYJOBPROP	2L
DM_RENDERCOMPLETE	0x0328	Drag Image Manipulation Flags	
DM_RENDERFILE	0x0322	DRG_BITMAP	0x00000002L
DM_RENDERPREPARE	0x0327	DRG_CLOSED	0x00000020L
GpiSet/QueryDrawingMode Constants		DRG_ICON	0x00000001L
DM_DRAW	1L	DRG_POLYGON	0x00000004L
DM_DRAWANDRETAIN	3L	DRG_STRETCH	0x00000008L
DM_ERROR	0L	DRG_TRANSPARENT	0x00000010L
DM_RETAIN	2L	GpiBox/GpiFullArc Fill Options	
Drag Error Return Values		DRO_FILL	1L
DME_IGNOREABORT	1		
DME_IGNORECONTINUE	2		
DME_REPLACE	3		
DME_RETRY	4		

PM Header Definition	Value	PM Header Definition	Value
DRO_OUTLINE	2L	EDI_LASTTABITEM	1
DRO_OUTLINEFILL	3L	EDI_NEXTGROUPITEM	6
Drag Object Type String Handles		EDI_NEXTTABITEM	2
DRT_ASM	"Assembler Code"	EDI_PREVGROUPITEM	7
DRT_BASIC	"BASIC Code"	EDI_PREVTABITEM	3
DRT_BINDATA	"Binary Data"	Entry Field Messages	
DRT_BITMAP	"Bitmap"	EM_CLEAR	0x0146
DRT_C	"C Code"	EM_COPY	0x0145
DRT_COBOL	"COBOL Code"	EM_CUT	0x0144
DRT_DLL	"Dynamic Link Library"	EM_PASTE	0x0147
DRT_DOSCMD	"DOS Command File"	EM_QUERYCHANGED	0x0140
DRT_EXE	"Executable"	EM_QUERYFIRSTCHAR	0x0148
DRT_FORTRAN	"FORTRAN Code"	EM_QUERY_READONLY	0x014a
DRT_ICON	"Icon"	EM_QUERYSEL	0x0141
DRT_LIB	"Library"	EM_SETFIRSTCHAR	0x0149
DRT_METAFILE	"Metafile"	EM_SETINSERTMODE	0x014c
DRT_OS2CMD	"OS/2 Command File"	EM_SET_READONLY	0x014b
DRT_PASCAL	"Pascal Code"	EM_SETSEL	0x0142
DRT_RESOURCE	"Resource File"	EM_SETTEXTLIMIT	0x0143
DRT_TEXT	"Plain Text"	Entry Field Notification Messages	
DRT_UNKNOWN	"Unknown"	EN_CHANGE	0x0004
Draw Text Codes		EN_INSERTMODETOGGLE	0x0040
DT_BOTTOM	0x0800	EN_KILLFOCUS	0x0002
DT_CENTER	0x0100	EN_MEMERROR	0x0010
DT_EXTERNALLEADING	0x0080	EN_OVERFLOW	0x0020
DT_ERASERECT	0x8000	EN_SCROLL	0x0008
DT_HALFTONE	0x1000	EN_SETFOCUS	0x0001
DT_LEFT	0x0000	GpiEqualRegion Return Codes	
DT_MNEMONIC	0x2000	EQRGN_ERROR	0L
DT_QUERYEXTENT	0x0002	EQRGN_EQUAL	2L
DT_RIGHT	0x0200	EQRGN_NOTEQUAL	1L
DT_STRIKEOUT	0x0020	Control Entry Field Styles	
DT_TEXTATTRS	0x0040	ES_ANY	0x00000000L
DT_TOP	0x0000	ES_AUTOSCROLL	0x00000004L
DT_UNDERSCORE	0x0010	ES_AUTOSIZE	0x000000200L
DT_VCENTER	0x0400	ES_AUTOTAB	0x00000010L
DT_WORDBREAK	0x4000	ES_CENTER	0x00000001L
Extended Attribute Flags		ES_COMMAND	0x00000040L
EAF_DEFAULTOWNER	0x0001	ES_DBCS	0x000002000L
EAF_REUSEICON	0x0004	ES_LEFT	0x00000000L
EAF_UNCHANGEABLE	0x0002	ES_MARGIN	0x00000008L
Enumerate Dialog Item Flags		ES_MIXED	0x00003000L
EDI_FIRSTGROUPITEM	4	ES_READONLY	0x00000020L
EDI_FIRSTTABITEM	0	ES_RIGHT	0x00000002L
EDI_LASTGROUPITEM	5	ES_SBCS	0x00001000L

PM Header Definition	Value	PM Header Definition	Value
ES_UNREADABLE	0x00000080L	FDS_PRELOAD_VOLINFO	0x00000020L
Focus Change Flags		FDS_SAVEAS_DIALOG	0x00000200L
FC_NOBRINGTOPFIRSTWINDOW	0x0002	File Dialog System Returned Attributes	
FC_NOBRINGTOTOP	0x0001		
FC_NOLOSEACTIVE	0x0008	FDS_EFSELECTION	0
FC_NOLOSEFOCUS	0x0002	FDS_LBSELECTION	1
FC_NOLOSESELECTION	0x0020	File Dialog System Error Return Codes	
FC_NOSETACTIVE	0x0004		
FC_NOSETFOCUS	0x0001	FDS_ERR_DIALOG_LOAD_ERROR	12
FC_NOSETSELECTION	0x0010	FDS_ERR DEALLOCATE_MEMORY	1
Frame Control Flags		FDS_ERR_DRIVE_ERROR	13
FCF_ACCELTABLE	0x00008000L	FDS_ERR_FILTER_TRUNC	2
FCF_AUTOICON	0x40000000L	FDS_ERR_INVALID_CUSTOM_HANDLE	11
FCF_BORDER	0x00000200L	FDS_ERR_INVALID_DIALOG	3
FCF_DBE_APPSTAT	0x80000000L	FDS_ERR_INVALID_DRIVE	4
FCF_DLGBORDER	0x00000100L	FDS_ERR_INVALID_FILTER	5
FCF_HORZSCROLL	0x00000080L	FDS_ERR_INVALID_PATHFILE	6
FCF_ICON	0x00004000L	FDS_ERR_INVALID_VERSION	10
FCF_MAXBUTTON	0x00000020L	FDS_ERR_OUT_OF_MEMORY	7
FCF_MENU	0x00000004L	FDS_ERR_PATH_TOO_LONG	8
FCF_MINBUTTON	0x00000010L	FDS_ERR_TOO_MANY_FILE_TYPES	9
FCF_MINMAX	0x00000030L	FDS_SUCCESSFUL	0
FCF_MOUSEALIGN	0x00040000L	Window Query Flags	
FCF_NOBYTEALIGN	0x00001000L		
FCF_NOMOVEWITHOWNER	0x00002000L	FF_ACTIVE	0x0002
FCF_PALETTE_HELP	0x00100000L	FF_DLGDISMISSED	0x0010
FCF_PALETTE_NORMAL	0x00080000L	FF_FLASHHILITE	0x0004
FCF_PALETTE_POPUPEVEN	0x00400000L	FF_FLASHWINDOW	0x0001
FCF_PALETTE_POPUPODD	0x00200000L	FF_NOACTIVATESWP	0x0080
FCF_SCREENALIGN	0x00020000L	FF_OWNERDISABLED	0x0020
FCF_SHELLPOSITION	0x000000400L	FF_OWNERHIDDEN	0x0008
FCF_SIZEBORDER	0x00000008L	FF_SELECTED	0x0040
FCF_STANDARD	0x0008CC3FL	GpiFloodFill Fill Options	
FCF_SYSMENU	0x00000002L		
FCF_SYSMODAL	0x00010000L	FF_BOUNDARY	0L
FCF_TASKLIST	0x00000800L	FF_SURFACE	1L
FCF_TITLEBAR	0x00000001L	Frame Information Flags	
FCF_VERTSCROLL	0x00000040L		
File Dialog System Flags		FI_ACTIVATEOK	0x00000004L
FDS_CENTER	0x00000001L	FI_FRAME	0x00000001L
FDS_CUSTOM	0x00000002L	FI_NOMOVEWITHOWNER	0x00000008L
FDS_ENABLEFILELB	0x00000800L	FI_OWNERHIDE	0x00000002L
FDS_FILTERUNION	0x00000004L	Frame Control IDs	
FDS_HELPBUTTON	0x00000008L		
FDS_INCLUDE_EAS	0x00000080L	FID_CLIENT	0x8008
FDS_MULTIPLESEL	0x00000400L	FID_DBE_APPSTAT	0x8010
FDS_OPEN_DIALOG	0x00000100L	FID_DBE_KBDSTAT	0x8011

PM Header Definition	Value	PM Header Definition	Value
FID_DBE_KKPOPUP	0x8013	Font Dialog Error Codes	
FID_DBE_PECIC	0x8012	FNTS_ERR_ALLOC_SHARED_MEM	4
FID_HORZSCROLL	0x8007	FNTS_ERR_DIALOG_LOAD_ERROR	12
FID_MENU	0x8005	FNTS_ERR_INVALID_DIALOG	3
FID_MINMAX	0x8004	FNTS_ERR_INVALID_PARM	5
FID_SYSMENU	0x8002	FNTS_ERR_INVALID_VERSION	10
FID_TITLEBAR	0x8003	FNTS_ERR_OUT_OF_MEMORY	7
FID_VERTSCROLL	0x8006	FNTS_SUCCESSFUL	0
Foreground Mixes		GpiCreateLogFont Return Codes	
FM_AND	6L	FONT_DEFAULT	1L
FM_DEFAULT	0L	FONT_MATCH	2L
FM_ERROR	(-1L)	GpiFillPath Modes	
FM_INVERT	12L	FPATH_ALTERNATE	0L
FM_LEAVEALONE	5L	FPATH_WINDING	2L
FM_MASKSRCNOT	8L	Frame Window Styles	
FM_MERGENOTSRC	15L	FS_ACCELTABLE	0x00000002L
FM_MERGESRCNOT	13L	FS_AUTOICON	0x00001000L
FM_NOTCOPYSRC	14L	FS_BORDER	0x000000100L
FM_NOTMASKSRC	16L	FS_DBE_APPSTAT	0x00008000L
FM_NOTMERGESRC	10L	FS_DLGBORDER	0x00000080L
FM_NOTXORSRC	11L	FS_ICON	0x00000001L
FM_ONE	17L	FS_MOUSEALIGN	0x00000040L
FM_OR	1L	FS_NOBYTEALIGN	0x00000010L
FM_OVERPAINT	2L	FS_NOMOVEWITHOWNER	0x00000020L
FM_SUBTRACT	7L	FS_SCREENALIGN	0x000000200L
FM_XOR	4L	FS_SHELLPOSITION	0x00000004L
FM_ZERO	9L	FS_SIZEBORDER	0x00000080L
Dialog Filter List Message String Identifiers		FS_STANDARD	0x0000000FL
FNTI_BITMAPFONT	0x0001	FS_SYSMODAL	0x000000040L
FNTI_FAMILYNAME	0x0010	FS_TASKLIST	0x00000008L
FNTI_STYLENAME	0x0020	GpiQueryFaceString Options	
FNTI_SYNTHESIZED	0x0004	FTYPE_ITALIC	0x0001
FNTI_VECTORFONT	0x0002	FTYPE_ITALIC_DONT_CARE	0x0002
Font Dialog Flags		FTYPE_OBLIQUE	0x0004
FNTF_PRINTERFONTSELECTED	2L	FTYPE_OBLIQUE_DONT_CARE	0x0008
FNTF_VIEWPRINTERFONTS	1L	FTYPE_ROUNDED	0x0010
Font Dialog Style Flags		FTYPE_ROUNDED_DONT_CARE	0x0020
FNTS_APPLYBUTTON	0x00000010L	GpiQueryFaceString WeightClass Options	
FNTS_CENTER	0x00000001L	FWEIGHT_BOLD	7L
FNTS_CUSTOM	0x00000002L	FWEIGHT_DONT_CARE	0L
FNTS_HELPBUTTON	0x00000008L	FWEIGHT_EXTRA_BOLD	8L
FNTS_MODELESS	0x00000040L	FWEIGHT_EXTRA_LIGHT	2L
FNTS_MULTIFONTSELECTION	0x00000004L		
FNTS_RESETBUTTON	0x00000020L		

PM Header Definition	Value	PM Header Definition	Value
FWEIGHT_LIGHT	3L	DevQueryHardcopyCaps Codes	
FWEIGHT_NORMAL	5L	HCAPS_CURRENT	1L
FWEIGHT_SEMI_BOLD	6L	HCAPS_SELECTABLE	2L
FWEIGHT_SEMI_LIGHT	4L	Initialization File Handles	
FWEIGHT_ULTRA_BOLD	9L	HINI_PROFILE	NULL
FWEIGHT_ULTRA_LIGHT	1L	HINI_SYSTEMPROFILE	-2L
GpiQueryFaceString WidthClass Options		HINI_USERPROFILE	-1L
FWIDTH_CONDENSED	3L	Hook Types	
FWIDTH_DONT_CARE	0L	HK_CHECKMSGFILTER	20
FWIDTH_EXPANDED	7L	HK_CODEPAGECHANGED	12
FWIDTH_EXTRA_CONDENSED	2L	HK_DESTROYWINDOW	16
FWIDTH_EXTRA_EXPANDED	8L	HK_FINDWORD	11
FWIDTH_NORMAL	5L	HK_HELP	5
FWIDTH_SEMI_CONDENSED	4L	HK_INPUT	1
FWIDTH_SEMI_EXPANDED	6L	HK_JOURNALRECORD	3
FWIDTH_ULTRA_CONDENSED	1L	HK_JOURNALPLAYBACK	4
FWIDTH_ULTRA_EXPANDED	9L	HK_LOADER	6
General GPI Return Values		HK_MSGCONTROL	8
GPI_ALTERROR	(-1L)	HK_MSGFILTER	2
GPI_ERROR	0L	HK_PLIST_ENTRY	9
GPI_OK	1L	HK_PLIST_EXIT	10
GpiCreatePS Associate Flags		HK_REGISTERUSERMSG	7
GPIA_ASSOC	0x4000L	HK_SENDMSG	0
GPIA_NOASSOC	0L	HK_WINDOWDC	15
GpiErrorSegmentData Error Context		Hook Help Modes	
GPIE_DATA	2L	HLP_M_FRAME	(-1)
GPIE_ELEMENT	1L	HLP_M_MENU	(-3)
GPIE_SEGMENT	0L	HLP_M_WINDOW	(-2)
GpiCreatePS Formats		Help Manager Messages	
GPIF_DEFAULT	0L	HM_ACTIONBAR_COMMAND	0x0233
GPIF_LONG	0x0200L	HM_CREATE_HELP_TABLE	0x0226
GPIF_SHORT	0x0100L	HM_DISMISS_WINDOW	0x0221
GpiCreatePS Presentation Space Formats		HM_DISPLAY_HELP	0x0222
GPIT_MICRO	0x1000L	HM_ERROR	0x022e
GPIT_NORMAL	0L	HM_EXT_HELP	0x0223
GpiResetPS Options		HM_EXT_HELP_UNDEFINED	0x0232
GRES_ALL	0x0004L	HM_GENERAL_HELP	0x0223
GRES_ATTRS	0x0001L	HM_GENERAL_HELP_UNDEFINED	0x0232
GRES_SEGMENTS	0x0002L	HM_HELP_CONTENTS	0x022b
		HM_HELP_INDEX	0x022a
		HM_HELPSUBITEM_NOT_FOUND	0x022f
		HM_INFORM	0x0234
		HM_INVALIDATE_DDF_DATA	0x023b
		HM_KEYS_HELP	0x022c

PM Header Definition	Value	PM Header Definition	Value
HM_LOAD_HELP_TABLE	0x0225	USERDATA	0x00f3
HM_NOTIFY	0x0242		Hittest Return Codes
HM_QUERY	0x023c	HT_DISCARD	(-2)
HM_QUERY_DDF_DATA	0x023a	HT_ERROR	(-3)
HM_QUERY_KEYS_HELP	0x0230	HT_NORMAL	0
HM_REPLACE_HELP_FOR_HELP	0x0229	HT_TRANSPARENT	(-1)
HM_REPLACE_USING_HELP	0x0229		Handle Specifications
HM_SET_ACTIVE_WINDOW	0x0224	HWND_BOTTOM	4
HM_SET_COVERPAGE_SIZE	0x023d	HWND_DESKTOP	1
HM_SET_HELP_LIBRARY_NAME	0x022d	HWND_OBJECT	2
HM_SET_HELP_WINDOW_TITLE	0x0227	HWND_THREADCAPTURE	5
HM_SET_OBJCOM_WINDOW	0x0238	HWND_TOP	3
HM_SET_SHOW_PANEL_ID	0x0228		Image Bundle Masks
HM_SET_USERDATA	0x0243	IBB_BACK_COLOR	0x0002L
HM_TUTORIAL	0x0231	IBB_BACK_MIX_MODE	0x0008L
HM_UPDATE_OBJCOM_WINDOW_CHAIN	0x0239	IBB_COLOR	0x0001L
		IBB_MIX_MODE	0x0004L
Help Manager Display Panel Constants		Dialog String Table IDs	
HM_RESOURCEID	0	IDS_FILE_ALL_FILES_SELECTOR	1000
HM_PANELNAME	1	IDS_FILE_BACK_CUR_PATH	1001
HM_PANELTYPE_NUMBER	0	IDS_FILE_BACK_PREV_PATH	1002
HM_PANELTYPE_NAME	1	IDS_FILE_BACK_SLASH	1003
CMIC_HIDE_PANEL_ID	0x0000	IDS_FILE_BAD_DRIVE_NAME	1100
CMIC_SHOW_PANEL_ID	0x0001	IDS_FILE_BAD_DRIVE_OR_PATH_NAME	1101
CMIC_TOGGLE_PANEL_ID	0x0002	IDS_FILE_BAD_FILE_NAME	1102
		IDS_FILE_BAD_FQF	1103
Help Manager Begin List Formatting Flags		IDS_FILE_BAD_NETWORK_NAME	1104
HMBT_NONE	1L	IDS_FILE_BAD_SUB_DIR_NAME	1105
HMBT_ALL	2L	IDS_FILE_BASE_FILTER	1004
HMBT_FIT	3L	IDS_FILE_BLANK	1005
		IDS_FILE_COLON	1006
Help Manager HMQW_VIEWPORT Values		IDS_FILE_DOT	1007
HMQVP_GROUP	0x0003	IDS_FILE_DRIVE_DISK_CHANGE	1120
HMQVP_NAME	0x0002	IDS_FILE_DRIVE_INVALID	1126
HMQVP_NUMBER	0x0001	IDS_FILE_DRIVE_LETTERS	1008
		IDS_FILE_DRIVE_LOCKED	1123
Help Manager HM_Query Constants		IDS_FILE_DRIVE_NO_SECTOR	1124
HMQW_ACTIVEVIEWPORT	0x000a	IDS_FILE_DRIVE_NOT_AVAILABLE	1106
HMQW_COVERPAGE	0x0001	IDS_FILE_DRIVE_NOT_READY	1122
HMQW_GROUP_VIEWPORT	0x00f1	IDS_FILE_DRIVE_SOME_ERROR	1125
HMQW_LIBRARY	0x0006	IDS_FILE_DUMMY_DRIVE	1021
HMQW_INDEX	0x0002	IDS_FILE_DUMMY_FILE_EXT	1020
HMQW_INSTANCE	0x0009	IDS_FILE_DUMMY_FILE_NAME	1019
HMQW_OBJCOM_WINDOW	0x0008	IDS_FILE_DUMMY_ROOT_DIR	1022
HMQW_RES_VIEWPORT	0x00f2		IDS_FILE_INVALID_CHARS
HMQW_SEARCH	0x0004		1027
HMQW_TOC	0x0003		
HMQW_VIEWPAGES	0x0005		
HMQW_VIEWPORT	0x0007		

PM Header Definition	Value	PM Header Definition	Value
IDS_FILE_FORWARD_SLASH	1011		Journal Notify Message Commands
IDS_FILE_FQFNAME_TOO_LONG	1107	JRN_PHYSKEYSTATE	0x00000002L
IDS_FILE_FWD_CUR_PATH	1009	JRN_QUEUESTATUS	0x00000001L
IDS_FILE_FWD_PREV_PATH	1010		WM_CHAR Field Bits
IDS_FILE_INSERT_DISK_NOTE	1127		
IDS_FILE_OK_WHEN_READY	1128	KC_ALT	0x0020
IDS_FILE_OPEN_DIALOG_NOTE	1108	KC_CHAR	0x0001
IDS_FILE_PARENT_DIR	1012	KC_COMPOSITE	0x0400
IDS_FILE_PATH_PTR	1023	KC_CTRL	0x0010
IDS_FILE_PATH_PTR2	1026	KC_DEADKEY	0x0200
IDS_FILE_PATH_TOO_LONG	1109	KC_DBCSRSRVD1	0x4000
IDS_FILE_Q_MARK	1013	KC_DBCSRSRVD2	0x8000
IDS_FILE_SAVEAS_DIALOG_NOTE	1110	KC_INVALIDCHAR	0x2000
IDS_FILE_SAVEAS_FILTER_TXT	1017	KC_INVALIDCOMP	0x0800
IDS_FILE_SAVEAS_FILENO_TXT	1018	KC_KEYUP	0x0040
IDS_FILE_SAVEAS_TITLE	1016	KC_LONEKEY	0x0100
IDS_FILE_SPLAT	1014	KC_NONE	0x0000
IDS_FILE_SPLAT_DOT	1015	KC_PREVDOWN	0x0080
IDS_FILE_VOLUME_PREFIX	1024	KC_SCANCODEE	0x0004
IDS_FILE_VOLUME_SUFFIX	1025	KC_SHIFT	0x0008
IDS_FONT_BLANK	351	KC_TOGGLE	0x1000
IDS_FONT_COMBINED	357	KC_VIRTUALKEY	0x0002
IDS_FONT_DISP_ONLY	355		Line Bundle Mask Bits
IDS_FONT_KEY_SEP	354	LBB_BACK_COLOR	0x0002L
IDS_FONT_KEY_0	352	LBB_BACK_MIX_MODE	0x0008L
IDS_FONT_KEY_9	353	LBB_COLOR	0x0001L
IDS_FONT_OPTION0	376	LBB_END	0x0080L
IDS_FONT_OPTION1	377	LBB_GEOM_WIDTH	0x0020L
IDS_FONT_OPTION2	378	LBB_JOIN	0x0100L
IDS_FONT_OPTION3	379	LBB_MIX_MODE	0x0004L
IDS_FONT_POINT_SIZE_LIST	380	LBB_TYPE	0x0040L
IDS_FONT_PRINTER_ONLY	356	LBB_WIDTH	0x0010L
IDS_FONT_SAMPLE	350		Local Character Set ID Values
IDS_FONT_WEIGHT1	358	LCID_ALL	(-1L)
IDS_FONT_WEIGHT2	359	LCID_DEFAULT	0L
IDS_FONT_WEIGHT3	360	LCID_ERROR	(-1L)
IDS_FONT_WEIGHT4	361		Local Character Set ID Types
IDS_FONT_WEIGHT5	362	LCIDT_BITMAP	7L
IDS_FONT_WEIGHT6	363	LCIDT_FONT	6L
IDS_FONT_WEIGHT7	364		Logical Color Options
IDS_FONT_WEIGHT8	365	LCOL_OVERRIDE_DEFAULT_COLORS	0x0008L
IDS_FONT_WEIGHT9	366	LCOL_PURECOLOR	0x0004L
IDS_FONT_WIDTH1	367		
IDS_FONT_WIDTH2	368		
IDS_FONT_WIDTH3	369		
IDS_FONT_WIDTH4	370		
IDS_FONT_WIDTH5	371		
IDS_FONT_WIDTH6	372		
IDS_FONT_WIDTH7	373		
IDS_FONT_WIDTH8	374		
IDS_FONT_WIDTH9	375		

PM Header Definition	Value	PM Header Definition	Value
LCOL_REALIZABLE	0x0002L		Line Width Values
LCOL_REALIZED	0x0010L	LINETHICK_DEFAULT	0L
LCOL_RESET	0x0001L	LINETHICK_ERROR	(-1L)
		LINETHICK_NORMAL	0x00010000L
Logical Color Table Formats		LINETHICK_THICK	0x00020000L
LCOLF_CONSECRGB	2L		List Box Constants
LCOLF_DEFAULT	0L	LIT_CURSOR	(-4)
LCOLF_INDRGB	1L	LIT_END	(-1)
LCOLF_PALETTE	4L	LIT_ERROR	(-3)
LCOLF_RGB	3L	LIT_FIRST	(-1)
		LIT_MEMERROR	(-2)
Logical Color Options		LIT_NONE	(-1)
LCOLOPT_INDEX	0x0002L	LIT_SORTASCENDING	(-2)
LCOLOPT_REALIZED	0x0001L	LIT_SORTDESCENDING	(-3)
			List Box Messages
Hook Loader Context Codes		LM_DELETEALL	0x016e
LHK_DELETELIB	2	LM_DELETEITEM	0x0163
LHK_DELETEPROC	1	LM_INSERTITEM	0x0161
LHK_LOADLIB	4	LM_QUERYITEMCOUNT	0x0160
LHK_LOADPROC	3	LM_QUERYITEMHANDLE	0x016a
		LM_QUERYITEMTEXT	0x0168
Line Ending Values		LM_QUERYITEMTEXTLENGTH	0x0167
LINEEND_DEFAULT	0L	LM_QUERYSELECTION	0x0165
LINEEND_ERROR	(-1L)	LM_QUERYTOPINDEX	0x016d
LINEEND_FLAT	1L	LM_SEARCHSTRING	0x016b
LINEEND_ROUND	3L	LM_SELECTITEM	0x0164
LINEEND_SQUARE	2L	LM_SETITEMHANDLE	0x0169
		LM_SETITEMHEIGHT	0x016c
Line Join Styles		LM_SETITEMTEXT	0x0166
LINEJOIN_BEVEL	1L	LM_SETTOPINDEX	0x0162
LINEJOIN_DEFAULT	0L		List Box Notification Messages
LINEJOIN_ERROR	(-1L)	LN_KILLFOCUS	3
LINEJOIN_MITRE	3L	LN_ENTER	5
LINEJOIN_ROUND	2L	LN_SCROLL	4
		LN_SELECT	1
Line Type Styles		LN_SETFOCUS	2
LINETYPE_ALTERNATE	9L		List Box Styles
LINETYPE_DASHDOT	3L	LS_EXTENDEDSEL	0x00000010L
LINETYPE_DASHDOUBLEDOT	6L	LS_HORIZSCROLL	0x00000008L
LINETYPE_DEFAULT	0L	LS_MULTIPLESEL	0x00000001L
LINETYPE_DOT	1L	LS_NOADJUSTPOS	0x00000004L
LINETYPE_DOUBLEDOT	4L	LS_OWNERDRAW	0x00000002L
LINETYPE_ERROR	(-1L)		
LINETYPE_INVISIBLE	8L		
LINETYPE_LONGDASH	5L		
LINETYPE_SHORTDASH	2L		
LINETYPE_SOLID	7L		

PM Header Definition	Value	PM Header Definition	Value
Marker Symbols		MBB_BOX	0x0040L
MARKSYM_BLANK	64L	MBB_COLOR	0x0001L
MARKSYM_CROSS	1L	MBB_MIX_MODE	0x0004L
MARKSYM_DEFAULT	0L	MBB_SET	0x0010L
MARKSYM_DIAMOND	3L	MBB_SYMBOL	0x0020L
MARKSYM_DOT	9L	Message Box Return Codes	
MARKSYM_EIGHTPOINTSTAR	6L	MBID_ABORT	3
MARKSYM_ERROR	(-1L)	MBID_CANCEL	2
MARKSYM_PLUS	2L	MBID_ENTER	9
MARKSYM_SIXPOINTSTAR	5L	MBID_ERROR	0xffff
MARKSYM_SMALLLCIRCLE	10L	MBID_HELP	8
MARKSYM_SOLIDDIAMOND	7L	MBID_IGNORE	5
MARKSYM_SOLIDsquare	8L	MBID_NO	7
MARKSYM_square	4L	MBID_OK	1
Message Box Button/Icon Codes		MBID_RETRY	4
MB_ABORTRETRYIGNORE	0x0003	MBID_YES	6
MB_APPLMODAL	0x0000	Menu Item Attributes	
MB_CANCEL	0x0006	MIA_CHECKED	0x2000
MB_CRITICAL	0x0040	MIA_DISABLED	0x4000
MB_CUACRITICAL	0x0040	MIA_FRAMED	0x1000
MB_CUANOTIFICATION	0x0000	MIA_HILITED	0x8000
MB_CUAWARNING	0x0020	MIA_NODISMISS	0x0020
MB_DEFBUTTON1	0x0000	Menu Item Styles	
MB_DEFBUTTON2	0x0100	MIS_BITMAP	0x0002
MB_DEFBUTTON3	0x0200	MIS_BREAK	0x0400
MB_ENTER	0x0007	MIS_BREAKSEPARATOR	0x0800
MB_ENTERCANCEL	0x0008	MIS_BUTTONSEPARATOR	0x0200
MB_ERROR	0x0040	MIS_GROUP	0x1000
MB_HELP	0x2000	MIS_HELP	0x0080
MB_ICONASTERISK	0x0030	MIS_MULTIMENU	0x0020
MB_ICONHAND	0x0040	MIS_OWNERDRAW	0x0008
MB_ICONQUESTION	0x0010	MIS_SEPARATOR	0x0004
MB_ICONEXCLAMATION	0x0020	MIS_SINGLE	0x2000
MB_INFORMATION	0x0030	MIS_STATIC	0x0100
MB_MOVEABLE	0x4000	MIS_SUBMENU	0x0010
MB_NOICON	0x0000	MIS_SYSCOMMAND	0x0040
MB_OK	0x0000	MIS_TEXT	0x0001
MB_OKCANCEL	0x0001	Menu Item Codes	
MB_QUERY	0x0010	MIT_END	(-1)
MB_RETRYCANCEL	0x0002	MIT_ERROR	(-1)
MB_SYSTEMMODAL	0x1000	MIT_FIRST	(-2)
MB_WARNING	0x0020	MIT_LAST	(-3)
MB_YESNO	0x0004	MIT_MEMERROR	(-1)
MB_YESNOCANCEL	0x0005	MIT_NONE	(-1)
Marker Bundle Masks			
MBB_BACK_COLOR	0x0002L		
MBB_BACK_MIX_MODE	0x0008L		

PM Header Definition	Value	PM Header Definition	Value
Multi-Line Edit Window Format Flags		MLM_QUERYFIRSTCHAR	0x01d6
MLFFMTRECT_FORMATRECT	0x00000007L	MLM_QUERYFONT	0x01d0
MLFFMTRECT_LIMITHORZ	0x00000001L	MLM_QUERYFORMATRECT	0x01b3
MLFFMTRECT_LIMITVERT	0x00000002L	MLM_QUERYFORMATLINELENGTH	0x01c7
MLFFMTRECT_MATCHWINDOW	0x00000004L	MLM_QUERYFORMATTEXTLENGTH	0x01c8
Multi-Line Edit window Import/Export Format flags		MLM_QUERYIMPORTEXPORT	0x01df
MLFIE_CFTEXT	0	MLM_QUERYLINECOUNT	0x01bc
MLFIE_NOTRANS	1	MLM_QUERYLINELENGTH	0x01bf
MLFIE_RTF	3	MLM_QUERYREADONLY	0x01b9
MLFIE_WINFMT	2	MLM_QUERYSEL	0x01cb
Multi-Line Entry Window Margin Indicators		MLM_QUERYSELTEXT	0x01cc
MLFMARGIN_BOTTOM	0x0002	MLM_QUERYTABSTOP	0x01b7
MLFMARGIN_LEFT	0x0001	MLM_QUERYTEXTCOLOR	0x01d3
MLFMARGIN_RIGHT	0x0003	MLM_QUERYTEXTLENGTH	0x01c0
MLFMARGIN_TOP	0x0004	MLM_QUERYTEXTLIMIT	0x01b1
Multi-Line Entry Query Selection Flags		MLM_QUERYUNDO	0x01cd
MLFQS_ANCHORSEL	3	MLM_QUERYWRAP	0x01b5
MLFQS_CURSORSEL	4	MLM_RESETUNDO	0x01cf
MLFQS_MAXSEL	2	MLM_SEARCH	0x01de
MLFQS_MINMAXSEL	0	MLM_SETBACKCOLOR	0x01d4
MLFQS_MINSEL	1	MLM_SETCOLOR	0x01bb
Multi-Line Entry Window Search Style Flags		MLM_SETFIRSTCHAR	0x01d7
MLFSEARCH_CASESENSITIVE	0x00000001L	MLM_SETFONT	0x01d1
MLFSEARCH_CHANGEALL	0x00000004L	MLM_SETFORMATRECT	0x01b2
MLFSEARCH_SELECTMATCH	0x00000002L	MLM_SETIMPORTEXPORT	0x01c2
Multi-Line Entry Window Messages		MLM_SET_READONLY	0x01b8
MLM_CHARFROMLINE	0x01bd	MLM_SETSEL	0x01ca
MLM_CLEAR	0x01db	MLM_SETTABSTOP	0x01b6
MLM_COPY	0x01d9	MLM_SETTEXTCOLOR	0x01d2
MLM_CUT	0x01d8	MLM_SETTEXTLIMIT	0x01b0
MLM_DELETE	0x01c6	MLM_SETWRAP	0x01b4
MLM_DISABLEREFRESH	0x01dd	MLM_UNDO	0x01ce
MLM_ENABLEREFRESH	0x01dc	Multi-Line Entry Notification Messages	
MLM_EXPORT	0x01c4	MLN_CHANGE	0x0007
MLM_FORMAT	0x01c1	MLN_CLPBDFAIL	0x000f
MLM_IMPORT	0x01c3	MLN_HSCROLL	0x0006
MLM_INSERT	0x01c9	MLN_KILLFOCUS	0x0009
MLM_LINEFROMCHAR	0x01be	MLN_MARGIN	0x000a
MLM_PASTE	0x01da	MLN_MEMERROR	0x000c
MLM_QUERYBACKCOLOR	0x01d5	MLN_OVERFLOW	0x0001
MLM_QUERYCHANGED	0x01ba	MLN_PIXHORZOVERFLOW	0x0002
MLS_BORDER	0x00000002L	MLN_PIXVERTOVERFLOW	0x0003

PM Header Definition	Value
MLS_DISABLEUNDO	0x00000040L
MLS_HSCROLL	0x00000008L
MLS_IGNORETAB	0x00000020L
MLS_READONLY	0x00000010L
MLS_VSCROLL	0x00000004L
MLS_WORDWRAP	0x00000001L
Menu Messages	
MM_DELETEITEM	0x0181
MM_ENDMENUMODE	0x0186
MM_INSERTITEM	0x0180
MM_ISITEMVALID	0x0193
MM_ITEMIDFROMPOSITION	0x0190
MM_ITEMPOSITIONFROMID	0x018f
MM_QUERYITEM	0x0182
MM_QUERYITEMATTR	0x0191
MM_QUERYITEMCOUNT	0x0184
MM_QUERYITEMRECT	0x0194
MM_QUERYITEMTEXT	0x018b
MM_QUERYITEMTEXTLENGTH	0x018c
MM_QUERYSELITEMID	0x018a
MM_REMOVEITEM	0x0188
MM_SELECTITEM	0x0189
MM_SETITEM	0x0183
MM_SETITEMATTR	0x0192
MM_SETITEMHANDLE	0x018d
MM_SETITEMTEXT	0x018e
MM_STARTMENUMODE	0x0185
Menu Styles	
MS_ACTIONBAR	0x00000001L
MS_TITLEBUTTON	0x00000002L
MS_VERTICALFLIP	0x00000004L
DevOpenDC Type Values	
OD_DIRECT	5L
OD_INFO	6L
OD_MEMORY	8L
OD_METAFILE	7L
OD_METAFILE_NOQUERY	9L
OD_QUEUED	2L
Help Manager HM_NOTIFY Codes	
OPEN_COVERPAGE	0x0001
OPEN_HISTORY	0x0006
OPEN_INDEX	0x0005
OPEN_LIBRARY	0x0008
OPEN_PAGE	0x0002
OPEN_SEARCH_HIT_LIST	0x0007
PM Header Definition	Value
OPEN_TOC	0x0004
SWAP_PAGE	0x0003
Basic Pattern Symbols	
PATSYM_BLANK	64L
PATSYM_DEFAULT	0L
PATSYM_DENSE1	1L
PATSYM_DENSE2	2L
PATSYM_DENSE3	3L
PATSYM_DENSE4	4L
PATSYM_DENSE5	5L
PATSYM_DENSE6	6L
PATSYM_DENSE7	7L
PATSYM_DENSE8	8L
PATSYM_DIAGHATCH	19L
PATSYM_DIAG1	11L
PATSYM_DIAG2	12L
PATSYM_DIAG3	13L
PATSYM_DIAG4	14L
PATSYM_ERROR	(-1L)
PATSYM_HALFTONE	17L
PATSYM_HATCH	18L
PATSYM_HORIZ	10L
PATSYM_NOSHADE	15L
PATSYM_SOLID	16L
PATSYM_VERT	9L
Palette Color Flags	
PC_EXPLICIT	0x02
PC_NOCOLLAPSE	0x04
PC_RESERVED	0x01
GpiSetPickApertureSize Options	
PICKAP_DEFAULT	0L
PICKAP_REC	2L
GpiCorrelateChain Correlation Types	
PICKSEL_ALL	1L
PICKSEL_VISIBLE	0L
WinPeekMsg Constants	
PM_NOREMOVE	0x0000
PM_REMOVE	0x0001
GpiPlayMetaFile Option Array Index Values	
PMF_COLORREALIZABLE	7
PMF_COLORTABLES	6

PM Header Definition	Value	PM Header Definition	Value
PMF_DEFAULTS	8	PRQ_PRIORITY_PARMNUM	2
PMF_DELETEOBJECTS	9	PRQ_PROCESSOR_PARMNUM	6
PMF_LCIDS	3	PRQ_REMOTE_COMPUTER_PARMNUM	15
PMF_LOADTYPE	1	PRQ_REMOTE_QUEUE_PARMNUM	16
PMF_RESET	4	PRQ_SEPARATOR_PARMNUM	5
PMF_RESOLVE	2	PRQ_STARTTIME_PARMNUM	3
PMF_SEGBASE	0	PRQ_TYPE_PARMNUM	10
PMF_SUPPRESS	5	PRQ_UNTILTIME_PARMNUM	4
Presentation Parameters		DosPrintDestSetInfo Parameter Number	
PP_ACTIVECOLOR	18L	PRD_COMMENT_PARMNUM	7
PP_ACTIVECOLORINDEX	19L	PRD_DRIVER_DATA_PARMNUM	11
PP_ACTIVETEXTBGNDCOLOR	24L	PRD_DRIVERS_PARMNUM	8
PP_ACTIVETEXTBGNDCOLORINDEX	25L	PRD_LOGADDR_PARMNUM	3
PP_ACTIVETEXTFGNDCOLOR	22L	PRD_TIMEOUT_PARMNUM	10
PP_ACTIVETEXTFGNDCOLORINDEX	23L	Print Destination Control Options	
PP_BACKGROUNDCOLOR	3L	PRD_DELETE	0
PP_BACKGROUNDCOLORINDEX	4L	PRD_PAUSE	1
PP_BORDERCOLOR	13L	PRD_CONT	2
PP_BORDERCOLORINDEX	14L	PRD_RESTART	3
PP_DISABLEDBACKGROUNDCOLOR	11L	Status Mask Flags of PRDINFO	
PP_DISABLEDBACKGROUNDCOLORINDEX	12L	PRD_ACTIVE	0
PP_DISABLEDFOREGROUNDCOLOR	9L	PRD_PAUSED	1
PP_DISABLEDFOREGROUNDCOLORINDEX	10L	PRD_STATUS_MASK	0x0003
PP_FONTHANDLE	16L	GpiPtInRegion Return Codes	
PP_FONTNAMESIZE	15L	PRGN_ERROR	0L
PP_FOREGROUNDCOLOR	1L	PRGN_INSIDE	2L
PP_FOREGROUNDCOLORINDEX	2L	PRGN_OUTSIDE	1L
PP_HILITEBACKGROUNDCOLOR	7L	GpiSet/QueryAttributes Bundle Codes	
PP_HILITEBACKGROUNDCOLORINDEX	8L	PRIM_AREA	4L
PP_HILITEFOREGROUNDCOLOR	5L	PRIM_CHAR	2L
PP_HILITEFOREGROUNDCOLORINDEX	6L	PRIM_IMAGE	5L
PP_INACTIVECOLOR	20L	PRIM_LINE	1L
PP_INACTIVECOLORINDEX	21L	PRIM_MARKER	3L
PP_INACTIVETEXTBGNDCOLOR	28L	Print Job Priority	
PP_INACTIVETEXTBGNDCOLORINDEX	29L	PRJ_MAX_PRIORITY	99
PP_INACTIVETEXTFGNDCOLOR	26L	PRJ_MIN_PRIORITY	1
PP_INACTIVETEXTFGNDCOLORINDEX	27L	PRJ_NO_PRIORITY	0
PP_RESERVED	17L	DosPrintJobSetInfo Parameter Number	
PP_SHADOW	30L	PRJ_COMMENT_PARMNUM	11
PP_USER	0x8000L		
DosPrintQSetInfo Parameter Number			
PRQ_COMMENT_PARMNUM	9		
PRQ_DESTINATIONS_PARMNUM	7		
PRQ_DRIVERDATA_PARMNUM	14		
PRQ_DRIVERNAME_PARMNUM	13		
PRQ_MAXPARMNUM	16		
PRQ_PARMS_PARMNUM	8		
PRQ_PRINTERS_PARMNUM	12		

PM Header Definition	Value	PM Header Definition	Value
PRJ_DATATYPE_PARMNUM	4		Print Queue Priority
PRJ_DOCUMENT_PARMNUM	12	PRQ_DEF_PRIORITY	5
PRJ_DRIVERDATA_PARMNUM	18	PRQ_MAX_PRIORITY	1
PRJ_JOBFILEINUSE_PARMNUM	7	PRQ_MIN_PRIORITY	9
PRJ_MAXPARMNUM	18	PRQ_NO_PRIORITY	0
PRJ_NOTIFYNAME_PARMNUM	3		
PRJ_PARMS_PARMNUM	5		
PRJ_POSITION_PARMNUM	6		
PRJ_PRIORITY_PARMNUM	14		
PRJ_PROCPARMS_PARMNUM	16		
PRJ_STATUSCOMMENT_PARMNUM	13		
PRJINFO Status Field Masks			
PRJ_COMPLETE	0x0004	PRQ_ACTIVE	0
PRJ_DELETED	0x8000	PRQ_ERROR	2
PRJ_DESTCRTCHG	0x0400	PRQ_STATUS_MASK	3
PRJ_DESTFORMCHG	0x0200	PRQ_PAUSED	1
PRJ_DESTNOPAPER	0x0100	PRQ_PENDING	3
PRJ_DESTOFFLINE	0x0020		
PRJ_DESTPAUSED	0x0040		
PRJ_DESTPENCHG	0x0800		
PRJ_DEVSTATUS	0x0ffc		
PRJ_ERROR	0x0010		
PRJ_INTERV	0x0008		
PRJ_JOBFILEINUSE	0x4000		
PRJ_NOTIFY	0x0080		
PRJ_QSTATUS	0x0003		
PRJ_QSTATUS Values			
PRJ_QS_PAUSED	1		
PRJ_QS_PRINTING	3		
PRJ_QS_QUEUED	0		
PRJ_QS_SPOOLING	2		
PM Group Program Categories			
PROG_DEFAULT	0		
PROG_DLL	6		
PROG_FULLSCREEN	1		
PROG_GROUP	5		
PROG_PDD	8		
PROG_PM	3		
PROG_REAL	4		
PROG_VDD	9		
PROG_VDM	4		
PROG_WINDOWABLEVIO	2		
PROG_WINDOWEDVDM	7		
PROG_WINDOW_AUTO	12		
PROG_WINDOW_PROT	11		
PROG_WINDOW_REAL	10		
Font Measurement Specifications			
PU_ARBITRARY	0x0004L		
PU_HIENGLISH	0x0018L		
PU_LOENGLISH	0x0014L		
PU_HIMETRIC	0x0010L		
PU_LOMETRIC	0x000CL		
PU_PELS	0x0008L		
PU_TWIPS	0x001CL		
Popup Menu Flags			
PU_HCONSTRAIN	0x0002		
PU_KEYBOARD	0x0200		

PM Header Definition	Value	PM Header Definition	Value
PU_MOUSEBUTTON1	0x0040	QPDAT_COMMENT	4
PU_MOUSEBUTTON1DOWN	0x0008	QPDAT_DATA_TYPE	3
PU_MOUSEBUTTON2	0x0080	QPDAT_DOC_NAME	8
PU_MOUSEBUTTON2DOWN	0x0010	QPDAT_DRIVER_DATA	2
PU_MOUSEBUTTON3	0x0100	QPDAT_DRIVER_NAME	1
PU_MOUSEBUTTON3DOWN	0x0018	QPDAT_JOBID	11
PU_NONE	0x0000	QPDAT_NET_PARAMS	7
PU_POSITIONONITEM	0x0001	QPDAT_PROC_PARAMS	5
PU_SELECTITEM	0x0020	QPDAT_QUEUE_NAME	9
PU_VCONSTRAIN	0x0004	QPDAT_SPL_PARAMS	6
GpiPtVisible Return Codes		QPDAT_TOKEN	10
		Presentation Parameters Flags	
PVIS_ERROR	0L	QPF_ID1COLORINDEX	0x0002
PVIS_INVISIBLE	1L	QPF_ID2COLORINDEX	0x0004
PVIS_VISIBLE	2L	QPF_NOINHERIT	0x0001
GpiQueryColorData Return Codes		QPF_PURERGBCOLOR	0x0008
QCD_LCT_FORMAT	0L	QPF_VALIDFLAGS	0x000F
QCD_LCT_HIINDEX	2L	Font Query Options	
QCD_LCT_LOINDEX	1L	QUERY_PUBLIC_FONTS	0x0001
QCD_LCT_OPTIONS	3L	QUERY_PRIVATE_FONTS	0x0002
Query Font Options		Queue Status Constants	
QF_NO_DEVICE	0x0008L	QS_KEY	0x0001
QF_NO_GENERIC	0x0004L	QS_MOUSE	0x0006
QF_PRIVATE	0x0002L	QS_MOUSEBUTTON	0x0002
QF_PUBLIC	0x0001L	QS_MOUSEMOVE	0x0004
Query Font Actions		QS_PAINT	0x0010
QFA_ERROR	(-1L)	QS_POSTMSG	0x0020
QFA_PRIVATE	2L	QS_SEM1	0x0040
QFA_PUBLIC	1L	QS_SEM2	0x0080
Query Focus Chain Constants		QS_SEM3	0x0100
QFC_ACTIVE	0x0002	QS_SEM4	0x0200
QFC_FRAME	0x0003	QS_SENDMSG	0x0400
QFC_NEXTINCHAIN	0x0001	QS_TIMER	0x0008
QFC_PARTOFCHAIN	0x0005	Window Query Codes	
QFC_SELECTACTIVE	0x0004	QW_BOTTOM	3
GpiQueryLogColorTable Return Codes		QW_FRAMEOWNER	8
QLCT_ERROR	(-1L)	QW_NEXT	0
QLCT_RGB	(-2L)	QW_NEXTTOP	6
PQPOPENDATA Block Elements		QW_OWNER	4
QPDAT_ADDRESS	0	QW_PARENT	5
		QW_PREV	1
		QW_PREVTOP	7
		QW_TOP	2

PM Header Definition	Value	PM Header Definition	Value
Window Word Indices (ULong)		RGN_RECT	2L
QWL_DEFBUTTON	0x0040	GpiBitBlt Raster Operations	
QWL_HHEAP	0x0004	ROP_DSTINVERT	0x0055L
QWL_HMQ	(-4)	ROP_MERGECOPY	0x00C0L
QWL_HWNDFOCUSSAVE	0x0018	ROP_MERGEPAINT	0x00BBL
QWL_MIN	(-6)	ROP_NOTSRCCOPY	0x0033L
QWL_PFEPBBLK	0x004c	ROP_NOTSRCERASE	0x0011L
QWP_PFNWP	(-3)	ROP_ONE	0x0FFL
QWL_PSSCBLK	0x0048	ROP_PATCOPY	0x00F0L
QWL_PSTATBLK	0x0050	ROP_PATINVERT	0x005AL
QWL_RESERVED	(-5)	ROP_PATPAINT	0x00FBBL
QWL_STYLE	(-2)	ROP_SRCAND	0x0088L
QWL_USER	0	ROP_SRCCOPY	0x00CC0L
Window Word Indices (UShort)		ROP_SRCERASE	0x0044L
QWS_CXRESTORE	0x0010	ROP_SRCINVERT	0x0066L
QWS_CYRESTORE	0x0012	ROP_SRCPAINT	0x00EEL
QWS_FLAGS	0x0008	ROP_ZERO	0x0000L
QWS_ID	(-1)	GpiRectInRegion Return Codes	
QWS_MIN	(-1)	RRGN_ERROR	0L
QWS_RESULT	0x000a	RRGN_INSIDE	3L
QWS_USER	0	RRGN_OUTSIDE	1L
QWS_XMINIMIZE	0x0014	RRGN_PARTIAL	2L
QWS_YMINIMIZE	0x0016	GpiRectVisible Return Codes	
QWS_XRESTORE	0x000c	RVIS_ERROR	0L
QWS_YRESTORE	0x000e	RVIS_INVISIBLE	1L
GpiQueryRegionRects Return Region Data		RVIS_PARTIAL	2L
RECTDIR_LFRT_BOTTOP	3L	RVIS_VISIBLE	3L
RECTDIR_LFRT_TOPBOT	1L	Scroll Bar Commands	
RECTDIR_RTLF_BOTTOP	4L	SB_ENDSCROLL	7
RECTDIR_RTLF_TOPBOT	2L	SB_LINEDOWN	2
RGB Colors		SB_LINELEFT	1
RGB_BLACK	0x00000000L	SB_LINERIGHT	2
RGB_BLUE	0x000000FFL	SB_LINEUP	1
RGB_CYAN	0x0000FFFFL	SB_PAGEDOWN	4
RGB_ERROR	(-255L)	SB_PAGELEFT	3
RGB_GREEN	0x0000FF00L	SB_PAGERIGHT	4
RGB_PINK	0x00FF00FFL	SB_PAGEUP	3
RGB_RED	0x00FF0000L	SB_SLIDERPOSITION	6
RGB_WHITE	0x00FFFFFFL	SB_SLIDERTRACK	5
RGB_YELLOW	0x00FFFF00L	Scroll Bar Messages	
Region Type Return Codes		SBM_QUERYPOLY	0x01a2
RGN_COMPLEX	3L	SBM_QUERYRANGE	0x01a3
RGN_ERROR	0L	SBM_SETPOS	0x01a1
RGN_NULL	1L		

PM Header Definition	Value	PM Header Definition	Value
SBM_SETSCROLLBAR	0x01a0	SBS_AUTOTRACK	4L
SBM_SETTHUMBSIZE	0x01a6	SBS_HORZ	0L
System Bitmaps		SBS_THUMBSIZE	2L
SBMP_BTNCORNERS	8	SBS_VERT	1L
SBMP_CHECKBOXES	48	System Command Values	
SBMP_CHILDSYSMENU	29	SC_APPMENU	0x8006
SBMP_CHILDSYSMENUDEP	34	SC_BEGINDRAG	0x8020
SBMP_COMBODOWN	47	SC_CLOSE	0x8004
SBMP_DRIVE	15	SC_CONTEXTHELP	0x8025
SBMP_FILE	16	SC_CONTEXTMENU	0x8024
SBMP_FOLDER	17	SC_DBE_FIRST	0x8018
SBMP_MAXBUTTON	27	SC_DBE_LAST	0x801F
SBMP_MAXBUTTONDEP	32	SC_ENDDRAG	0x8021
SBMP_MENUATTACHED	23	SC_HELPXENDED	0x8014
SBMP_MENUCHECK	6	SC_HELPINDEX	0x8013
SBMP_MINBUTTON	26	SC_HELPKEYS	0x8012
SBMP_MINBUTTONDEP	31	SC_MAXIMIZE	0x8003
SBMP_OLD_CHILDSYSMENU	12	SC_MINIMIZE	0x8002
SBMP_OLD_CHECKBOXES	7	SC_MOVE	0x8001
SBMP_OLD_MAXBUTTON	10	SC_NEXT	0x8005
SBMP_OLD_MINBUTTON	9	SC_NEXTFRAME	0x8009
SBMP_OLD_RESTOREBUTTON	11	SC_NEXTWINDOW	0x8010
SBMP_OLD_SBDNARROW	3	SC_OPEN	0x8023
SBMP_OLD_SBLFARROW	5	SC_RESTORE	0x8008
SBMP_OLD_SBRGARROW	4	SC_SELECT	0x8022
SBMP_OLD_SBUPARROW	2	SC_SIZE	0x8000
SBMP_OLD_SYSMENU	1	SC_SWITCHPANELIDS	0x8015
SBMP_PROGRAM	22	SC_SYSMENU	0x8007
SBMP_RESTOREBUTTON	28	SC_TASKMANAGER	0x8011
SBMP_RESTOREBUTTONDEP	33	SC_TEXTEDIT	0x8026
SBMP_SBDNARROW	36	GpiSetClipPath Modes	
SBMP_SBDNARROWDEP	40	SCP_ALTERNATE	0L
SBMP_SBDNARROWDIS	44	SCP_AND	4L
SBMP_SBLFARROW	37	SCP_RESET	0L
SBMP_SBLFARROWDEP	41	SCP_WINDING	2L
SBMP_SBLFARROWDIS	45	Slider Control Ownerdraw Attributes	
SBMP_SBRGARROW	38	SDA_BACKGROUND	0x0003
SBMP_SBRGARROWDEP	42	SDA_RIBBONSTRIP	0x0001
SBMP_SBRGARROWDIS	46	SDA_SLIDERARM	0x0004
SBMP_SBUPARROW	35	SDA_SLIDERSHAFT	0x0002
SBMP_SBUPARROWDEP	39	Desktop Flags	
SBMP_SBUPARROWDIS	43	SDT_CENTER	0x0020L
SBMP_SIZEBOX	24	SDT_DESTROY	0x0001L
SBMP_SYSMENU	25	SDT_LOADFILE	0x0080L
SBMP_SYSMENUDEP	30		
SBMP_TREEMINUS	19		
SBMP_TREEPLUS	18		
Scroll Bar Styles			
SBS_AUTOSIZE	0x2000L		

PM Header Definition	Value	PM Header Definition	Value
SDT_NOBKND	0x0002L	SLM_SETSCALETEXT	0x0370
SDT_PATTERN	0x0010L	SLM_SETSLIDERINFO	0x0371
SDT_RETAIN	0x0040L	SLM_SETTICKSIZE	0x0372
SDT_SCALE	0x0008L		
SDT_TILE	0x0004L		
		Slider Control Notification Messages	
GpiSet/QueryStopDraw Constants		SLN_CHANGE	1
SDW_ERROR	-1L	SLN_KILLFOCUS	4
SDW_OFF	0L	SLN_SETFOCUS	3
SDW_ON	1L	SLN_SLIDERTRACK	2
		Slider Control Styles	
GpiSet/QueryEditMode Edit Modes		SLS_BOTTOM	0x00000002L
SEGEM_ERROR	0L	SLS_BUTTONSBOTTOM	0x00000010L
SEGEM_INSERT	1L	SLS_BUTTONSLEFT	0x00000010L
SEGEM_REPLACE	2L	SLS_BUTTONSRIGHT	0x00000020L
		SLS_BUTTONSTOP	0x00000020L
WinSetErrInfo Constants		SLS_CENTER	0x00000000L
SEI_ARGCOUNT	0x0004	SLS_HOMEBOTTOM	0x00000000L
SEI_BREAKPOINT	0x8000	SLS_HOMELEFT	0x00000000L
SEI_DBGRSRVD	0x1000	SLS_HOMERIGHT	0x00000200L
SEI_DEBUGONLY	0xEF00	SLS_HOMETOP	0x00000200L
SEI_DOSERROR	0x0008	SLS_HORIZONTAL	0x00000000L
SEI_MSGSTR	0x0010	SLS_LEFT	0x00000002L
SEI_NOBEEP	0x4000	SLS_OWNERDRAW	0x00000040L
SEI_NOPROMPT	0x2000	SLS_PRIMARYSCALE1	0x00000000L
SEI_REGISTERS	0x0002	SLS_PRIMARYSCALE2	0x00000400L
SEI_RESERVED	0x0FE0	SLS_READONLY	0x00000080L
SEI_STACKTRACE	0x0001	SLS_RIBBONSTRIP	0x00000100L
		SLS_RIGHT	0x00000004L
Program Structure Visibility/Protect Flags		SLS_SNAPTOINCREMENT	0x00000008L
SHE_INVISIBLE	0x01	SLS_TOP	0x00000004L
SHE_PROTECTED	0x02	SLS_VERTICAL	0x00000001L
SHE_RESERVED	0xFF		
SHE_UNPROTECTED	0x00	Static Control Messages	
SHE_VISIBLE	0x00	SM_QUERYHANDLE	0x0101
		SM_SETHANDLE	0x0100
Slider Control Error Message IDs			
SLDERR_INVALID_PARAMETERS	-1	Slider Control Attributes	
		SMA_INCREMENTVALUE	0x0001
Slider Control Messages		SMA_RANGEVALUE	0x0000
SLM_ADDDETENT	0x0369	SMA_SCALE1	0x0001
SLM_QUERYDETENTPOS	0x036a	SMA_SCALE2	0x0002
SLM_QUERYSCALETEXT	0x036b	SMA_SETALLTICKS	0xFFFF
SLM_QUERYSLIDERINFO	0x036c	SMA_SHAFTDIMENSIONS	0x0000
SLM_QUERYTICKPOS	0x036d	SMA_SHAFTPOSITION	0x0001
SLM_QUERYTICKSIZE	0x036e	SMA_SLIDERARMDIMENSIONS	0x0002
SLM_REMOVEDETENT	0x036f	SMA_SLIDERARMPOSITION	0x0003

PM Header Definition	Value	PM Header Definition	Value
Message Mode Constants		SPBS_READONLY	0x00000002L
SMD_DELAYED	0x0001	SPBS_SERVANT	0x00000000L
SMD_IMMEDIATE	0x0002	Printer Type Flags	
Message Interest Constants		SPL_PR_DIRECT_DEVICE	0x00000002
SMI_AUTODISPATCH	0x0008	SPL_PR_LOCAL_ONLY	0x00000100
SMI_INTEREST	0x0002	SPL_PR_QUEUE	0x00000001
SMI_NOINTEREST	0x0001	SPL_PR_QUEUED_DEVICE	0x00000004
SMI_RESET	0x0004	Spooler Message Box Error Data	
SMIM_ALL	0x0EFF	SPLDATA_CARTCHGREQD	0x0004
Spin Button Control Messages		SPLDATA_DATAERROR	0x0010
SPBM_OVERRIDESETLIMITS	0x200	SPLDATA_FORMCHGREQD	0x0002
SPBM_QUERYLIMITS	0x201	SPLDATA_OTHER	0x8000
SPBM_QUERYVALUE	0x205	SPLDATA_PENCHGREQD	0x0008
SPBM_SETARRAY	0x206	SPLDATA_PRINTERJAM	0x0001
SPBM_SETCURRENTVALUE	0x208	SPLDATA_UNEXPECTERROR	0x0020
SPBM_SETLIMITS	0x207	SplMessageBox Error Information	
SPBM_SETMASTER	0x209	SPLINFO_QPERROR	0x0001
SPBM_SETTEXTLIMIT	0x202	SPLINFO_DDERROR	0x0002
SPBM_SPINDOWN	0x204	SPLINFO_SPLERROR	0x0004
SPBM_SPINUP	0x203	SPLINFO_OTHERERROR	0x0080
Spin Button Notification Messages		SPLINFO_INFORMATION	0x0100
SPBN_CHANGE	0x20D	SPLINFO_WARNING	0x0200
SPBN_DOWNARROW	0x20B	SPLINFO_ERROR	0x0400
SPBN_ENDSPIN	0x20C	SPLINFO_SEVERE	0x0800
SPBN_KILLFOCUS	0x20F	SPLINFO_USERINTREQD	0x1000
SPBN_SETFOCUS	0x20E	SplQpControl Control Codes	
SPBN_UPARROW	0x20A	SPLC_ABORT	1
Spin Button Query Flags		SPLC_CONTINUE	3
SPBQ_ALWAYSUPDATE	1	SPLC_PAUSE	2
SPBQ_DONOTUPDATE	3	System Pointers	
SPBQ_UPDATEIFVALID	0	SPTR_APPICON	10
Spin Button Styles		SPTR_ARROW	1
SPBS_ALLCHARACTERS	0x00000000L	SPTR_BANGICON	14
SPBS_FASTSPIN	0x00000100L	SPTR_CPTR	14
SPBS_JUSTCENTER	0x0000000CL	SPTR_FILE	19
SPBS_JUSTDEFAULT	0x00000000L	SPTR_FOLDER	20
SPBS_JUSTLEFT	0x00000008L	SPTR_HANDICON	13
SPBS_JUSTRIGHT	0x00000004L	SPTR_ICONERROR	13
SPBS_MASTER	0x00000010L	SPTR_ICONINFORMATION	11
SPBS_NOBORDER	0x00000020L	SPTR_ICONQUESTION	12
SPBS_NUMERICONLY	0x00000001L	SPTR_ICONWARNING	14
SPBS_PADWITHZEROS	0x00000080L	SPTR_ILLEGAL	18
		SPTR_MOVE	5

PM Header Definition	Value	PM Header Definition	Value
SPTR_MULTFILE	21	SV_CTIMERS	47
SPTR_NOTEICON	11	SV_CURSORLEVEL	45
SPTR_PROGRAM	22	SV_CURSORRATE	9
SPTR_QUESICON	12	SV_CXALIGN	49
SPTR_SIZE	4	SV_CYALIGN	50
SPTR_SIZENESW	7	SV_CXBORDER	26
SPTR_SIZENS	9	SV_CYBORDER	27
SPTR_SIZENWSE	6	SV_CXBYTEALIGN	49
SPTR_SIZEWE	8	SV_CYBYTEALIGN	50
SPTR_TEXT	2	SV_CXCHORD	71
SPTR_WAIT	3	SV_CYCHORD	72
Static Control Styles		SV_CXDBLCLK	2
SS_AUTOSIZE	0x0040L	SV_CYDBLCLK	3
SS_BITMAP	0x0004L	SV_CXDLGFRAME	28
SS_BKGNDFRAME	0x000aL	SV_CYDLGFRAME	29
SS_BKGNDRECT	0x0007L	SV_CXFULLSCREEN	36
SS_FGNDFRAME	0x0008L	SV_CYFULLSCREEN	37
SS_FGNDRECT	0x0005L	SV_CXHSCROLLARROW	25
SS_GROUPBOX	0x0002L	SV_CXHSLIDER	32
SS_HALFTONEFRAME	0x0009L	SV_CXICON	38
SS_HALFTONERECT	0x0006L	SV_CYICON	39
SS_ICON	0x0003L	SV_CXICONCONTEXTWIDTH	68
SS_SYSICON	0x000bL	SV_CXMINMAXBUTTON	33
SS_TEXT	0x0001L	SV_CYMINMAXBUTTON	34
GpiQueryTextBox Array Indices		SV_CXMOTION	73
TXTBOX_BOTTOMLEFT	1L	SV_CYMOTION	74
TXTBOX_BOTTOMRIGHT	3L	SV_CXSCREEN	20
TXTBOX_CONCAT	4L	SV_CYSCREEN	21
TXTBOX_TOPLEFT	0L	SV_CXSIZEBORDER	4
TXTBOX_TOPRIGHT	2L	SV_CYSIZEBORDER	5
System Value Identities		SV_CYHSCROLL	23
SV_ALARM	6	SV_CYMENU	35
SV_ALTMNEMONIC	66	SV_CXPOINTER	40
SV_ANIMATION	90	SV_CYPOINTER	41
SV_ANIMATIONSLEPTIME	93	SV_CYTITLEBAR	30
SV_ANIMATIONSPEED	92	SV_CYVSCROLLARROW	24
SV_BEGINDRAG	75	SV_CYVSLIDER	31
SV_BEGINDRAGKB	82	SV_DEBUG	42
SV_CHORDTIME	70	SV_DBLCLKTIME	1
SV_CICONTEXTLINES	69	SV_ENDDRAG	76
SV_CONTEXTHELP	80	SV_ENDDRAGKB	83
SV_CONTEXTHELPKB	87	SV_ERRORDURATION	18
SV_CONTEXTMENU	79	SV_ERRORFREQ	15
SV_CONTEXTMENUKB	86	SV_EXTRAKEYBEEP	57
SV_CMOUSEBUTTONS	43	SV_FIRSTSCROLLRATE	10
SV_CPOINTERBUTTONS	43	SV_INSERTMODE	59
SV_CSYSVALUES	98	SV_KBDALTERED	96
		SV_KBTRANSLATEFIRST	82
		SV_KBTRANSLATELAST	88
		SV_MENUROLLOWDDELAY	64
		SV_MENUROLLUPDELAY	65
		SV_MONOICONS	95
		SV_MOUSEPRESENT	48

PM Header Definition	Value	PM Header Definition	Value
SZDDESYS_ITEM_HELP	"Help"	TF_GRID	0x0020
SZDDESYS_ITEM_ITEMFORMATS	"ItemFormats"	TF_LEFT	0x0001
SZDDESYS_ITEM_PROTOCOLS	"Protocols"	TF_MOVE	0x000F
SZDDESYS_ITEM_RESTART	"Restart"	TF_PARTINBOUNDARY	0x0200
SZDDESYS_ITEM_RTNMSG	"ReturnMessage"	TF_RIGHT	0x0004
SZDDESYS_ITEM_STATUS	"Status"	TF_SETPOINTERPOS	0x0010
SZDDESYS_ITEM_SYSITEMS	"Sysitems"	TF_STANDARD	0x0040
SZDDESYS_ITEM_SECURITY	"Security"	TF_TOP	0x0002
SZDDESYS_ITEM_TOPICS	"Topics"	TF_VALIDATETRACKRECT	0x0100
Clipboard Format Strings		Timer ID Flags	
SZFMT_BITMAP	"#2"	TID_CURSOR	0xffff
SZFMT_CPTEXT	"Codepage Text"	TID_FLASHWINDOW	0xffffd
SZFMT_DIB	"Dib"	TID_SCROLL	0xffffe
SZFMT_DIF	"Dif"	TID_USERMAX	0x7fff
SZFMT_DSPBITMAP	"#4"	GpiSetSegmentTransformMatrix Transformation Types	
SZFMT_DSPMETAFILE	"#6"	TRANSFORM_ADD	1L
SZFMT_DSPMETAFILEPICT	"DspMetaFilePict"	TRANSFORM_PREEMPT	2L
SZFMT_DSPTEXT	"#3"	TRANSFORM_REPLACE	0L
SZFMT_LINK	"Link"	Value Set Control Item Attributes	
SZFMT_METAFILE	"#5"	VIA_BITMAP	0x0001
SZFMT_METAFILEPICT	"MetaFilePict"	VIA_COLORINDEX	0x0010
SZFMT_OEMTEXT	"OemText"	VIA_DISABLED	0x0040
SZFMT_OWNERDISPLAY	"OwnerDisplay"	VIA_DRAGGABLE	0x0080
SZFMT_PALETTE	"#9"	VIA_DROPONABLE	0x0100
SZFMT_SYLK	"Sylk"	VIA_ICON	0x0002
SZFMT_TEXT	"#1"	VIA_OWNERDRAW	0x0020
SZFMT_TIFF	"Tiff"	VIA_RGB	0x0008
Text Character Alignments		VIA_TEXT	0x0004
TA_BASE	0x0400	Virtual Key Codes	
TA_BOTTOM	0x0500	VK_ALT	0x0B
TA_CENTER	0x0003	VK_ALTPGRAF	0x0C
TA_HALF	0x0300	VK_BACKSPACE	0x05
TA_LEFT	0x0002	VK_BACKTAB	0x07
TA_NORMAL_HORIZ	0x0001	VK_BREAK	0x04
TA_NORMAL_VERT	0x0100	VK_BUTTON1	0x01
TA_RIGHT	0x0004	VK_BUTTON2	0x02
TA_STANDARD_HORIZ	0x0005	VK_BUTTON3	0x03
TA_STANDARD_VERT	0x0600	VK_CAPSLOCK	0x0E
TA_TOP	0x0200	VK_CTRL	0x0A
Title Bar Messages		VK_DBCSFIRST	0x0080
TBM_QUERYHILITE	0x01e4	VK_DBCSLAST	0x00ff
TBM_SETHILITE	0x01e3	VK_DELETE	0x1B
Window Tracking Options		VK_DOWN	0x18
TF_ALLINBOUNDARY	0x0080	VK_END	0x13
TF_BOTTOM	0x0008		

PM Header Definition	Value	PM Header Definition	Value
VK_ENTER	0x1E	VM_SETITEM	0x0351
VK_ESC	0x0F	VM_SETITEMATTR	0x0352
VK_HOME	0x14	VM_SETMETRICS	0x0353
VK_INSERT	0x1A	Value Set Control Message Parameter Attributes	
VK_LEFT	0x15		
VK_MENU	0x29	VMA_ITEMSIZE	0x0001
VK_NEWLINE	0x08	VMA_ITEMSPACING	0x0002
VK_NUMLOCK	0x1D	Value Set Notification Messages	
VK_PAGEDOWN	0x12		
VK_PAGEUP	0x11	VN_DRAGLEAVE	122
VK_PAUSE	0x0D	VN_DRAGOVER	123
VK_PRINTSCRN	0x19	VN_DROP	124
VK_RIGHT	0x17	VN_DROPHHELP	125
VK_SCROLLLOCK	0x1C	VN_ENTER	121
VK_SHIFT	0x09	VN_HELP	129
VK_SPACE	0x10	VN_INITDRAG	126
VK_SYSRQ	0x1F	VN_KILLFOCUS	128
VK_TAB	0x06	VN_SELECT	120
VK_UP	0x16	VN_SETFOCUS	127
VK_USERFIRST	0x0100	Value Set Control Styles	
VK_USERLAST	0x01ff		
VK_F1	0x20	VS_BITMAP	0x0001
VK_F2	0x21	VS_ICON	0x0002
VK_F3	0x22	VS_TEXT	0x0004
VK_F4	0x23	VS_RGB	0x0008
VK_F5	0x24	VS_COLORINDEX	0x0010
VK_F6	0x25	VS_BORDER	0x0020
VK_F7	0x26	VS_ITEMBORDER	0x0040
VK_F8	0x27	VS_SCALEBITMAPS	0x0080
VK_F9	0x28	VS_RIGHTTOLEFT	0x0100
VK_F10	0x29	Alarm Codes	
VK_F11	0x2A		
VK_F12	0x2B	WA_CWINALARMS	3
VK_F13	0x2C	WA_ERROR	2
VK_F14	0x2D	WA_NOTE	1
VK_F15	0x2E	WA_WARNING	0
VK_F16	0x2F	Control Window Class Names	
VK_F17	0x30	WC_APPSTAT	0xffff0010L
VK_F18	0x31	WC_BUTTON	0xffff0003L
VK_F19	0x32	WC_CNRTREE	0xffff0091L
VK_F20	0x33	WC_COLORMSAMPLE	0xffff0025L
VK_F21	0x34	WC_COMBOBOX	0xffff0002L
VK_F22	0x35	WC_CONTAINER	0xffff0021L
VK_F23	0x36	WC_DBE_KKPOPUP	0xffff0013L
VK_F24	0x37	WC_ENTRYFIELD	0xffff0006L
Value Set Control Messages		WC_FONTSAMPLE	0xffff0026L
VM_QUERYITEM	0x034c	WC_FRAME	0xffff0001L
VM_QUERYITEMATTR	0x034d		
VM_QUERYMETRICS	0x034e		
VM_QUERYSELECTEDITEM	0x034f		
VM_SELECTITEM	0x0350		

PM Header Definition	Value	PM Header Definition	Value
WC_KBDSTAT	0xfffff0011L		
WC_LISTBOX	0xfffff0007L		
WC_MENU	0xfffff0004L		
WC_MLE	0xfffff000aL		
WC_NOTEBOOK	0xfffff0024L		
WC_PECIC	0xfffff0012L		
WC_SCROLLBAR	0xfffff0008L		
WC_SLIDER	0xfffff0022L		
WC_SPINBUTTON	0xfffff0020L		
WC_STATIC	0xfffff0005L		
WC_TITLEBAR	0xfffff0009L		
WC_VALUESET	0xfffff0023L		
Dynamic Data Exchange Messages			
WM_DDE_ACK	0x00A2		
WM_DDE_ADVISE	0x00A4		
WM_DDE_DATA	0x00A3		
WM_DDE_EXECUTE	0x00A7		
WM_DDE_FIRST	0x00A0		
WM_DDE_INITIATE	0x00A0		
WM_DDE_INITIATEACK	0x00A9		
WM_DDE_LAST	0x00AF		
WM_DDE_POKE	0x00A6		
WM_DDE_REQUEST	0x00A1		
WM_DDE_TERMINATE	0x00A8		
WM_DDE_UNADVISE	0x00A5		
Window Parameter Flags			
WPM_CBCTLDATA	0x0010		
WPM_CBPRESPARAMS	0x0020		
WPM_CCHTEXT	0x0008		
WPM_CTLDATA	0x0002		
WPM_PRESPARAMS	0x0004		
WPM_TEXT	0x0001		
Window Styles			
WS_ANIMATE	0x00400000L		
WS_CLIPCHILDREN	0x20000000L		
WS_CLIPSIBLINGS	0x10000000L		
WS_DISABLED	0x40000000L		
WS_GROUP	0x00010000L		
WS_MAXIMIZED	0x00800000L		
WS_MINIMIZED	0x01000000L		
WS_MULTISELECT	0x00040000L		
WS_PARENTCLIP	0x08000000L		
WS_SAVEBITS	0x04000000L		
WS_SYNCPAINT	0x02000000L		
WS_TABSTOP	0x00020000L		
WS_VISIBLE	0x80000000L		

Existing 16-Bit Function	New 32-Bit Function
<p>DosPrintDestAdd DosPrintDestControl DosPrintDestDel DosPrintDestEnum DosPrintDestGetInfo DosPrintDestSetInfo DosPrintDriverEnum DosPrintJobContinue DosPrintJobDelete DosPrintJobEnum DosPrintJobGetId DosPrintJobGetInfo DosPrintJobPause DosPrintJobSetInfo DosPrintPortEnum DosPrintQAdd DosPrintQContinue DosPrintQDel DosPrintQEnum DosPrintQGetInfo DosPrintQPause DosPrintProcessorEnum DosPrintQPurge DosPrintQSetInfo GpiRealizeColorTable GpiUnRealizeColorTable PrfAddProgram PrfChangeProgram PrfCreateGroup PrfDestroyGroup PrfQueryDefinitions PrfQueryProgramCategory</p>	<p>DdfBeginList DdfBitmap DdfEndList DdfHyperText DdfInform DdfInitialize DdfListItem DdfMetafile DdfPara DdfSetColor DdfSetFont DdfSetFontStyle DdfSetFormat DdfSetTextAlign DdfText SplCreateDevice SplControlDevice SplDeleteDevice SplEnumDevice SplQueryDevice SplSetDevice SplEnumDriver SplReleaseJob SplDeleteJob SplEnumJob SplQueryJobId SplQueryJob SplHoldJob SplSetJobInfo SplEnumPort SplCreateQueue SplReleaseQueue SplDeleteQueue SplEnumQueue SplQueryQueue SplHoldQueue SplEnumQueueProcessor SplPurgeQueue SplSetQueue</p>

Existing 16-Bit Function	New 32-Bit Function
PrfQueryProgramHandle PrfQueryProgramTitles PrfRemoveProgram	
WinAllocMem WinAvailMem WinCreateGroup WinCreateHeap	
WinDestroyHeap	WinCreateObject WinDeregisterObjectClass
WinFreeMem WinInstStartApp	WinDestroyObject WinEnumObjectClass WinFreeFileIcon
WinLockHeap WinLockWindow WinLockWindowUpdate WinQueryDefinition WinQueryProfileData	WinLoadFileIcon
WinQueryProfileInt WinQueryProfileSize WinQueryProfileString WinQueryProgramTitles WinQueryWindowLockCount	
WinReallocMem	WinQueryClassThunkProc WinQueryWindowMode WinQueryWindowThunkProc
WinTerminateApp WinWriteProfileData WinWriteProfileString	WinRegisterObjectClass WinReplaceObjectClass WinRestoreWindowPos WinSetClassThunkProc WinSetFileIcon WinSetObjectData WinSetWindowThunkProc WinShutDownSystem WinStoreWindowPosition

Existing 16-Bit Function	New 32-Bit Function
BadDynLink	
DosAllocHuge	DosAcknowledgeSignalException DosAddMuxWaitSem
DosAllocSeg DosAllocShrSeg DosBeep DosBufReset DosCallBack DosCallNmPipe	DosAllocMem DosAllocSharedMem
DosCaseMap DosChDir DosChgFilePtr DosCLIAccess DosClose	DosBeep DosResetBuffer
DosCloseQueue DosCloseSem	DosCallNPIPE DosCancelLockRequest DosMapCase DosSetCurrentDir DosSetFilePtr
DosConnectNmPipe DosCopy DosCreateCSAlias	DosClose DosCloseEventSem DosCloseMutexSem DosCloseMuxWaitSem DosCloseQueue
DosCreateQueue DosCreateSem DosCreateThread DosCWait DosDelete	DosCloseVDD DosConnectNPIPE DosCopy
DosDevConfig DosDevIOCtl DosDevIOCtl2 DosDisconnectNmPipe DosDupHandle DosEditName DosEnterCritSec	DosCreateEventSem DosCreateMutexSem DosCreateMuxWaitSem DosCreateQueue
DosEnumAttribute DosErrClass	DosCreateThread DosWaitChild DosDelete DosDeleteMuxWaitSem DosDevConfig DosDevIOCtl DosDevIOCtl DosDisconnectNPIPE DosDupHandle DosEditName DosEnterCritSec DosEnterMustComplete DosEnumAttribute DosErrClass

Existing 16-Bit Function	New 32-Bit Function
DosError	DosError
DosExecPgm	DosExecPgm
DosExit	DosExit
DosExitCritSec	DosExitCritSec
DosExitList	DosExitList
DosFileIO	DosExitMustComplete.
DosFileLocks	DosSetFileLocks
DosFindClose	DosFindClose
DosFindFirst	DosFindFirst
DosFindNext	DosFindNext
DosFindNotifyClose	
DosFindNotifyFirst	
DosFindNotifyNext	
DosFreeModule	DosFreeMem
DosFreeResource	DosFreeModule
DosFreeSeg	DosFreeResource
DosFSAttach	DosFSAttach
DosFSCtl	DosFSCtl
DosFSRamSemClear	
DosFSRamSemRequest	
DosGetCP	DosQueryCP
DosGetCtryInfo	DosQueryCtryInfo
DosGetDateTime	DosGetDateTime
DosGetDBCSEv	DosQueryDBCSEnv
DosGetCollate	DosQueryCollate
DosGetEnv	
DosGetHugeShift	
DosGetInfoSeg	DosGetInfoBlocks
DosGetMachineMode	
DosGetMessage	DosGetNamedSharedMem
DosGetModHandle	DosGetMessage
DosGetModName	DosQueryModuleHandle
DosGetPID	DosQueryModuleName
DosGetPPID	DosGetInfoBlocks
DosGetProcAddr	DosGetPPID
DosGetPrtv	DosQueryProcAddr
DosGetResource	DosGetInfoBlocks
DosGetResource2	DosGetResource
DosGetShrSeg	DosGetResource
DosGetSeg	DosGetSharedMem
DosGetVersion	
DosGiveSeg	

Existing 16-Bit Function	New 32-Bit Function
DosHoldSignal	DosGiveSharedMem
DosInsMessage	DosInsertMessage
DosKillProcess	DosKillProcess
DosLoadModule	DosKillThread
DosLockSeg	DosLoadModule
DosMkDir	DosCreateDir
DosMakePipe	DosCreatePipe
DosMakeNmPipe	DosCreateNPipe
DosMemAvail	
DosMove	DosMove
DosMuxSemWait	
DosNewSize	DosSetFileSize
DosOpen	DosOpen
DosOpen2	DosOpen
	DosOpenEventSem
DosOpenQueue	DosOpenMutexSem
DosOpenSem	DosOpenMuxWaitSem
	DosOpenQueue
DosPeekNmPipe	DosOpenVDD
DosPeekQueue	DosPeekNPipe
DosPhysicalDisk	DosPeekQueue
DosPortAccess	DosPhysicalDisk
DosPTrace	DosPostEventSem
DosPurgeQueue	DosDebug
DosPutMessage	DosPurgeQueue
DosQAppType	DosPutMessage
DosQCurDir	DosQueryAppType
DosQCurDisk	DosQueryCurrentDir
DosQFHandState	DosQueryCurrentDisk
DosQFileInfo	DosQueryDOSProperty
DosQ FileMode	DosQueryFHState
DosQFSAttach	DosQueryFileInfo
DosQFSInfo	
DosQHandType	DosQueryFSAttach
DosQNnmPHandState	DosQueryFSInfo
DosQNnmPipeInfo	DosQueryHType
DosQNnmPipeSemState	DosQueryNPHState
DosQPathInfo	DosQueryNPipeInfo
DosQSysInfo	DosQueryNPipeSemState
	DosQueryPathInfo
	DosQueryProcType
	DosQuerySysInfo

Existing 16-Bit Function	New 32-Bit Function
DosQueryQueue	DosQueryEventSem DosQueryMem DosQueryMessageCP DosQueryMutexSem DosQueryMuxWaitSem DosQueryQueue DosQueryResourceSize DosQueryVerify DosRaiseException DosRawReadNpipe DosRawWriteNpipe DosRead
DosQVerify	DosRead
DosRawReadNmPipe DosRawWriteNmPipe DosRead DosReadAsync DosReadQueue DosReallocHuge DosReallocSeg	DosReadQueue
DosResumeThread	DosReleaseMutexSem DosRequestMutexSem DosResetEventSem DosResumeThread DosRequestVDD
DosRetForward DosRmDir DosR2StackRealloc DosScanEnv DosSearchPath DosSelectDisk DosSemClear DosSemRequest DosSemSet DosSemSetWait DosSemWait DosSendSignal	DosDeleteDir DosScanEnv DosSearchPath DosSetDefaultDisk
DosSetCP DosSetDateTime	DosSendSignalException DosSetDateTime DosSetDOSProperty DosSetExceptionHandler DosSetFHState DosSetFileInfo
DosSetFHandState DosSetFileInfo DosSet FileMode DosSetFSInfo	DosSetFSInfo DosSetKBDSigFocus DosSetMaxFH DosSetRelMaxFH DosSetMem
DosSetMaxFH	

Existing 16-Bit Function	New 32-Bit Function
<p>DosSetNmPHandInfo DosSetNmPipeSem DosSetPathInfo DosSetProcCP DosSetPrty</p> <p>DosSetVec DosSetVerify DosSizeSeg DosSleep DosSelectSession</p> <p>DosSetSession DosShutDown DosSMRegisterDD DosStartSession DosStopSession DosSubAlloc DosSubFree DosSubSet</p> <p>DosSuspendThread DosTimerAsync DosTimerStart DosTimerStop DosTransactNmPipe DosUnLockSeg</p> <p>DosWaitNmPipe</p> <p>DosWrite DosWriteAsync DosWriteQueue</p>	<p>DosSetNPHState DosSetNPipeSem DosSetPathInfo DosSetProcessCP DosSetPriority DosSetSignalException DosSetSignalExceptionFocus</p> <p>DosSetVerify</p> <p>DosSleep DosSelectSession DosSetExceptionHandler DosSetSession DosShutDown</p> <p>DosStartSession DosStopSession DosSubAllocMem DosSubFreeMem DosSubSetMem DosSubUnSetMem DosSuspendThread DosAsyncTimer DosStartTimer DosStopTimer DosTransactNPipe</p> <p>DosUnSetExceptionHandler DosUnWindException DosWaitEventSem DosWaitMuxWaitSem DosWaitNPipe DosWaitThread DosWrite</p> <p>DosWriteQueue</p>

Index

A

Accelerator keys 140 - 143
ACCELTABLE statement 142 - 143
BEGIN and END statements 143
Key options 142
Load and memory options 142
KEYVALUE statement 142
Table 141
Address space
 Compatibility region 29
 Shared memory 30
 System region 30

B

BM-QUERYCHECK 248, 249, 259
BM-SETCHECK 248, 249, 255 - 256
BN-CLICKED 248, 249, 259, 260
Bounding rectangle 129 - 130
BUILD.CMD 72
Buttons
 Checkboxes 202, 203, 248 - 249
 Processing data from 258 - 259
 Push buttons 202, 203
 Radio buttons 202, 203, 247 - 248
 Processing data from 260
 User buttons 202, 203

C

C Bindings
 Using 49 - 50

Calling Convention
 COBOL 50
Class
 Control window 17
 Defining the name 82, 102
 MainWndClass 87, 126
 Registering 18
 Styles 19, 88, 102
COBOL
 Adding CONFIG.SYS statements 62
 Call parameter passing 51 - 52, 102
 Compiler
 Command file 63 - 64
 Directives 65 - 66
 Definition of
 DWORD 50 - 51
 Pointer 50 - 51
 WORD 50 - 51
 ENTRY statement 99
 EVALUATE statement 100, 108, 127
 When phrase 100, 108, 125, 149, 162,
 174, 194, 301
 When other phrase 100, 108, 126, 149,
 162, 194, 301
 EXIT statement 101
 Extended compiler vs. COBOL/2 Bindings
 48
 Migrating current programs to Version 2.0
 38 - 41
 Presentation Manager requirements
 45 - 46
 Program
 Essential parts of 57 - 60
 Programming model compatibility 37

- Running current compilers under
 - Version 2.0 38
- Under Version 2.0 28
- Using with the Presentation Manager
 - 45 - 60
- Code page 221
- Coding Conventions 57
- Combo boxes 202, 203
- Communication
 - Interprocess 12 - 13
- CONFIG.SYS
 - Modifying 62
- Container controls 202, 203
- Control window
 - Classes 20
 - Grouping 251 - 253
 - SS_GROUPBOX 250 - 252
 - WS_GROUP 250 - 252
 - WS_TABSTOP 253
 - Handle of 165
- Conversion
 - 16-bit programs to 32-bit 40 - 41
 - Suggestions during 41
- CREATEPARAMS data structure 191, 203 - 206

D

- Define statement 117, 140, 158, 172
- Desktop 21
 - Measuring the size of 90
- Development environment
 - Setting up 61
- Dialogs
 - Common properties 133
 - CONTROL statement 185
 - Control messages 202 - 203
 - Control windows
 - Dialog Frame windows 200
 - Interactive Control windows 200
 - Static Control windows 200
 - Also see individual controls
 - Data passing 191, 203
 - Establishing addressability 204 - 206
 - Passing the structure pointer 204
 - Also see CREATEPARAMS structure
 - Device context 332
 - DevEscape call 333, 335, 341
 - DevOpenDataStructure 338, 339
 - DevOpenDC call 332
 - DevPostDeviceModes call 332, 336, 337 - 338
 - DevQueryCaps call 332, 339
 - DIALOG statement 184 - 185
 - DLGINCLUDE statement 182, 207
 - DLGTEMPLATE statement 182 - 184
 - Dynamic dialog boxes 180
 - Ending 195, 218
 - Generated dialog format 186 - 187
 - Initialization processing 212, 255
 - Interpreting results 196
 - Message flow 192 - 193
 - Modal vs. Modeless 177
 - Notification messages 202
 - Procedures, coding 193
 - PRESPARAM statement 185
 - Processing 216 - 218
 - Program Termination dialog 178 - 182
 - Required resources 181
 - RCINCLUDE statement 207
 - Returned data, processing 263 - 264
 - Setting default controls 255 - 258
 - Static dialog boxes 180
 - DosAllocMem call 227, 337
 - DosAllocSeg call 227, 337
 - DosFreeMem call 335, 342
 - DosFreeModule call 307, 320 - 321
 - DosFreeSeg call 335, 342
 - DosGetProcAddr call 307
 - DosLoadModule call 307, 319 - 320
 - DWORD (Double Word) 51
 - Dynamic Linking
 - Building a 314 - 315
 - Compiling and linking a 315
 - Creating
 - An import library 313 - 314
 - The Linker Response file 308 - 309, 314, 316
 - The Module Definition file 309 - 312, 314, 316
 - Libraries 12
 - Loading 307
 - Preload vs. Loadoncall 308
 - Printer Setup dialog DLL 318, 321
 - Calling the entry point 322
 - Public vs. Private libraries 306
 - Resource-only DLL 315 - 316
 - Building a 317
 - Resource stub program 315
 - Uses for 305

E

Editors
 Dialog 74, 180 - 182
 Font 75
 Icon 78, 152
 Suggestions for using 153
EM-QUERYCHANGED 262
EM-SETTEXTLIMIT 257
EN-CHANGE 261
 Ending A PM Program 97 - 99
EN-KILLFOCUS 261
 Entry fields 202, 203, 250 - 251
 Processing data from 260 - 262
 Styles 251
 Environment Division 79, 102
EXPORTS statement 310

F

Focus window 16
 Fonts
 A word about 221
 Current 228
 Font attributes 226 - 228, 236, 334
 Font face vs. font 222
 Image vs. outline 223 - 224
 Local character set ID 230, 237, 239, 243,
 244
 Measurement of 222
 Metrics information 227, 229 - 230, 234
 Metrics table 226, 233, 237
 Positioning text 241 - 242
 Printers' points 222
 Public vs. Private 223, 234
 Raster 223
 Resetting the current font 243
 Selecting a desired 226 - 228
 Serif vs. sans serif 222
 Testing for 231
 Twips 223
 Vector 224
Frame Create flags (FCF-) 20 - 21, 88, 103,
 117, 119, 156
Frame Identity values (FID-) 166 - 167

G

GpiCharString call 240, 335
GpiCharStringAt call 240, 242

GpiCharStringPos call 240
GpiCharStringPosAt call 240
GpiCreateLogFont call 226, 228, 238, 334
GpiCreatePS call 128, 333
GpiDeleteSetID call 230, 243 - 244
GpiDestroyPS call 128
GpiErase call 233
GpiQueryFontMetrics call 230, 239
GpiQueryFonts call 226, 227, 234, 333
GpiSetCharBox call 334
GpiSetCharMode call 334
GpiSet CharSet call 230, 238 - 239, 243, 334
GpiSetCurrentPos call 335

H

hab 81, 87, 88, 95, 97, 99, 102, 145, 282, 283,
 297
Handles
 Coding variables 81
 Filter 94
 Working with 56
Hierarchy
 Window 21 - 22
HINI-PROFILE 281
HINI-SYSTEMPROFILE 281, 283
HINI-USERPROFILE 281, 284
hmq 81, 87, 98, 102
hps 130, 132, 234, 237, 239, 242, 243, 244
hwnd 101, 130, 195, 196, 213, 215, 217, 256,
 257, 262, 265, 268, 292, 302
HWND-DESKTOP 81, 91, 102, 119, 146 -
 147, 160, 162, 301
hwndFocusWindow 162
hwndFrame 81, 90, 98, 102, 119, 167, 264,
 285, 298, 299, 300
hwndMenu 169
hwndPointer 162
HWND-TOP 81, 92, 102
HWND-BOTTOM 81, 92

I

Icons
 Adding to the Resource Script file 155
 Drawing densities 152
 Drawing suggestions 153
 ICON statement 155 - 156
Identification Division 79, 102
IMPLIB.EXE 313

IMPORTS statement 312
Initialization files
 Application (user) 278
 Processing 281
Application name 273 - 274
Data types used in 280
Default system (1.3) 274 - 277
Fonts 231
Key name 273 - 274
MAKEINI.EXE 280
Structure of 273 - 274
SYS_DLLS 306
System 271 - 272
Writing to 287 - 289
Initialization routine 58, 85
Intel
 Reverse order of values 54
I/O instruction privilege levels 10

L

LIBPATH statement 62
LIBRARY statement 310
Linkage Section 84 - 85, 104
Linker
 Command file 63
 Response file 66 - 68
Linking
 Static PM calls 55
List boxes 202, 203,
 Loading 214 - 216
LIT-END command 214, 215
LIT-SORTASCENDING command 214
LIT-SORTDESCENDING command 214
LM-DELETEALL message 212, 213 - 214
LM-INSERTITEM message 214, 215
LM-QUERYSELECTION message 206,
 216 - 217
LM-SELECTITEM message 215, 216
LN-ENTER 216, 218
LN-SELECT message 206, 216 - 218
Local-Storage Section 83 - 84, 104

M

MAKE.CMD file 63 - 65
MAKEINI.EXE 280
Memory
 Flat model 28
 LDT tiling 33

Objects 30
Pages 30
Segmented memory 28
Thunk layers 34
16-bit compatibility 33 - 35
Menus
 Adding pull-down 136, 170
 Attributes 116, 139, 168 - 169, 176
 Controls 202, 203
 Creating 114
 ENTRY statement 114
 MENUITEM 115, 136, 138 - 141, 170 - 171
 Style flags 114 - 115
 SUBMENU 136 - 140, 171
Message(s) 23
 A word about 121 - 124
 Checking for selected 125
 Control *See* Dialogs
 Filtering 95
 Flow of 25
 How they are generated 25
 Keyboard codes 121 - 122
 Notification *See* Dialogs
 Parameters
 Coding 80
 Definition of 24 - 25
Posting 123
Processing routine 60, 85, 94, 126
Recursion 124
Sending 123
Structure definition
 Linkage Section 24
 Working-Storage 24
Message box 134 - 136
 Button styles 135
 Icon styles 135
 Modality values 135
 Return codes 136
 Styles 134 - 135
MM-SETITEMATTR message 167 - 168, 176
Module Definition file 68 - 69
Mouse Pointer
 Adding code to the sample program 159
 Adding to the Resource Script file 158
 Checking for mouse movement 161
 Creating custom 297 - 299
 Functions 157
 Hand grabbing pointers 297, 298
 Pointer statement 158
Multiline entry fields 202, 203

Multitasking 10

N

Notebook controls 202, 203

O

ORDINALS statement 311

OS/2

Call-return interface 12

Version 2.0 27 - 44

COBOL under 28

Compatibility with Version 1.3
programs 36 - 38

Migrating current Programs to 38 - 41

Programming models 37

Running current compilers under 38

OS/2.H header file 112

OS/2.INI *See* initialization files

OS/2SYS.INI *See* initialization files

P

Page

Fault exception 33

Guard 33

Guard exception 33

Management 31 - 33

Page Table 31

Page Table Directory 31

Parameter passing

By Reference 51 - 52

By Value 51 - 52

Returning 52, 55

Pascal calling convention 12

Implementing 50 - 51, 102

Pipes 13

PM Call dialog 204 - 219

Pointer 51, 87, 103, 204 - 206

Posting a message 123

Presentation Manager

Environment 13 - 14

Flow of information 25

Messages 23

Windows 14 - 23

Presentation Parameters 244 - 246

Presentation spaces

A word about 128

Cached 129

Micro 127

Regular 127

With text 230

PrfCloseProfile call 281 - 288

PrfOpenProfile call 281 - 288

PrfQueryProfileData call 280

PrfQueryProfileInt call 286

PrfQueryProfileString call 280, 283 - 284,
286, 331

PrfWriteProfileData call 280, 288 - 289

PrfWriteProfileString call 280, 288, 307

Printing

Data flow

Basic printing 326

Direct printing 327

Queued printing 326

Printer and job properties 328 - 329

Printer Queues 331

The printer data stream

Raw (PM_Q_RAW) 330

Building a 336 - 342

Standard (PM_Q_STD) 330

Building a 331 - 335

The printer subsystem

The Kernel device drivers 325

The Printer device drivers 325, 331

The Queue Processor 324, 329, 336, 341

The User interface dialogs 324

Procedure Division 85 - 86, 104

Procedures

ChildWndProc 175

ENTRY statement 99, 107

EXIT statement 101, 108

MainWndProc 107, 126, 174

Pointer 87, 103

Registering 87

Window 60, 99 - 101

Process 10

Address space 29

Profiles *See* initialization files

Program-done flag 95, 106

Programming models

COBOL compatibility for 37 - 38

Mixed 16-bit 37

Mixed 32-bit 37

Pure 16-bit 37

Pure 32-bit 37

Protect-mode memory addressing 8

Prtsamp.DLL 307 *

Q

QMSG- 24, 80, 95, 97, 102
 Queues, memory 13
 QWL- 296
 QWS- 296

R

RCL
 Structure 104
 Referencing 130, 132
 Real-mode memory addressing 7
 Resource
 A word about 109
 Accelerator key table
 See Accelerator keys
 Accessing string data 144 - 145
 Advantages of using 110
 BEGIN and END statements 112,
 137 - 140
 Coding conventions 111
 Compiler command file 72
 Compiling the Script file 117
 Creating the script file 110, 111 - 117
 #define statement 117, 140, 158, 172
 Defining in the COBOL program 118
 Dialogs *See* Dialogs
 Header files 116, 158
 Icons *See* Icons
 Load and memory options 113 - 114
 Menus *See* Menus
 Modifying the script file 139, 170, 187,
 207, 253
 Resource file 70, 110
 Script file 70, 110
 String table *See* String table
 Type definitions 113
 Returning 52, 55

S

Sample program
 A word about 77 - 79
 Control messages within 165
 Error processing in 78
 Modifying 117 - 119, 129 - 132, 145 - 149,
 159, 172 - 176, 187, 208, 233 - 243, 254,
 282 - 289, 297 - 303, 322
 Scroll bars 202, 203

Sending a message 123
 Sessions 10
 Set Window Position flags (SWP-) 82, 92,
 103
 Shared Memory 12
 Slider controls 202, 203
 Spin buttons 202, 203
 SplQmClose call 342
 SplQmEndDoc call 341
 SplQmOpen call 338, 339 - 340
 SplQmStartDoc call 340
 SplQmWrite call 340 - 341
 Static controls 202, 203
 Storage values
 Intel reverse order 54
 String Table 143 - 144, 208
 BEGIN and END statements 144
 Special ASCII characters 144
 STRINGTABLE statement 143 - 144
 Strings
 Accessing resource file 144 - 145
 Null-terminator
 Defining 54
 Use of 52 - 53
 Removing 53
 Working-Storage buffers 145
 Styles
 Class 19 - 20
 Window 18 - 20
 Subclassing *See* Window Subclassing
 System values (SV-) 91, 103, 302

T

Termination routine 60, 86
 Threads 10
 Priorities 12
 Thunk layers
 Functions of 34 - 35
 Title-bar controls 202, 203
 Toolkit 73, 74, 75, 152
 Translation Lookaside Buffer 32

V

Value Set controls 202, 203
 Variables
 Initializing 55
 Virtual
 address space 9

Segmentation hardware 10

W

WinBeginEnumWindows call 265 - 266, 267 - 268
 WinBeginPaint call 129 - 130
 WinCreateDlg call 178
 WinCreateMsgQueue call 85, 86, 105
 WinCreateStdWindow call 85, 88, 105
 WinCreateWindow call 85, 89
 WinDefDlgProc call 193 - 195
 WinDefWindowProc call 100 - 101, 108, 292
 WinDestroyMsgQueue call 85, 98, 107
 WinDestroyWindow call 86, 98, 107
 WinDismissDlg call 195 - 196, 218
 WinDispatchMsg call 86, 96, 100, 107, 163 - 164
 WinDlgBox call 190 - 192
 WinEndEnumWindows call 265 - 266, 268
 WinEndPaint call 132
 WinFillRect call 130 - 132
 WinGetMsg call 86, 94, 106
 WinGetNextWindow call 165, 265 - 266, 268
 WinInitialize call 85, 86, 105
 WinInvalidateRegion call 265
 WinLoadDlg call 178
 WinLoadPointer call 159 - 160, 297
 WinLoadString call 145, 207
 WinMessageBox call 146 - 147
 WinPostMsg call 163 - 164, 189 - 190, 292
 WinPostQueueMsg call 163 - 164
 WinProcessDlg call 178
 WinQueryFocus call 161 - 162, 301
 WinQuerySysValue call 91, 105, 106, 233
 WinQueryWindow call 165
 WinQueryWindowPos call 294
 WinQueryWindowText call 263
 WinQueryWindowULong call 296, 301 - 302, 303
 WinQueryWindowUShort call 296
 WinRegisterClass call 85, 87, 105, 173, 295, 297
 WinSendDlgItemMsg call 206, 213, 215, 216, 217, 218, 255 - 256, 257
 WinSendMsg call 163 - 164, 169, 292
 WinSetPointer call 162
 WinSetPresParam call 246
 WinSetWindowPos call 94, 106
 WinSetText call 219, 264, 285

WinSetWindowULong call 300
 WinSubclassWindow call 293, 294, 298 - 299
 WinTerminate call 85, 98, 107
 WinUpper call 283
 WinWindowFromID call 165, 167, 262
 WinWindowFromPointer call 165
 Window
 A word about 14
 Child 172 - 176
 Classes 17 - 18
 Control 20
 Enumeration 265 - 268
 Focus 16
 Frame 16
 Handles 56
 Hierarchy 21 - 22
 Owner 23
 Positions 82
 Procedures 60, 87, 99 - 101
 Programming procedures for 46 - 47
 Registering 18
 Sizing and positioning 90 - 94
 Styles 18 - 20
 Subclassing 291 - 303
 Implementing 292
 Procedure 300 - 301
 Returning unwanted messages 292, 302 - 303
 Single window vs. entire class 293 - 294
 Words 204
 A word about 294 - 296
 Allocating 297
 Application defined 295
 Index into 295
 Index values 296
 Loading 300
 Uses for 294
 Z-order of 16
 Window Edit dialog box 253
 Coding the procedure for 258
 Resources for 254
 WindowProc 292, 297, 303
 WM-BUTTONxDBLCLK 122
 WM-BUTTONxDOWN 122
 WM-BUTTONxUP 122
 WM-COMMAND 122, 140, 142, 170, 174, 216, 218, 254
 Coding the processing routine 148 - 149
 Structure of 148
 WM-CONTROL 200, 202, 206, 216, 248, 249

WM-HELP 122, 140, 142
WM-HSCROLL 200
WM-INITDLG 204, 216, 295
WM-MAXMINFRAME 157
WM-MOUSEMOVE 122, 159, 161 - 162, 291,
 301, 302
WM-PAINT 126, 127, 147, 264
WM-QUIT 83, 86, 96, 103, 106, 189, 218
WM-SETFOCUS 125
WM-SYSCOMMAND 122, 140, 142
WM-VSCROLL 200
WORD 51
Working-Storage Section 80, 102 - 103
Workplace Shell 35 - 36

Z

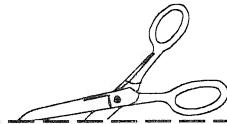
Z-order of windows 16

Ordering Sample Program Diskettes

As a convenience and learning aid you may order diskettes containing the current version of the sample programs contained in "The COBOL Presentation Manager Programming Guide". The files on these diskettes are shipped on an unsupported, as-is basis. Neither the Author nor Van Nostrand Reinhold assume any liability with respect to the use, accuracy, or fitness of information contained within these diskettes.

Send the order form with payment to:

**Dill & Company
P.O. Box 1331
Roanoke, Texas 76262**



**"The COBOL Presentation Manager Programming Guide"
Diskette Offer**

Name: _____

Company: _____

Address: _____

City: _____ State: _____ Zip Code: _____

2 - 3 1/2" 1.4 MBytes diskettes

Quantity _____ @ \$ 22.50 Total: \$ _____

Texas residents add 7.25% sales tax Tax: \$ _____

For postage and handling: P & H: \$ _____

U.S. orders add \$ 3.00

Foreign orders add \$ 5.00

Total Amount: \$ _____

Include a check or money order for the total amount, payable in U.S. dollars, to
David M. Dill.

Please allow 4 to 6 weeks for delivery within the U.S.

Take the direct—and cost-effective—route to programming OS/2® and Presentation Manager without C!

The COBOL Presentation Manager Programming Guide

David M. Dill

Moving applications from mainframes to OS/2? Contrary to popular myth, C is not the only language supported by the OS/2—Presentation Manager environment. COBOL is a real and viable alternative to writing in C and this definitive guide shows how to swiftly begin programming and converting your programs onto OS/2 2.0 without having to first learn C programming.

While providing COBOL programmers with a “fast path” to the first PM screen, this first-of-its-kind sourcebook is also an excellent tool for programmers to begin their Presentation Manager programming training, regardless of their language specialty. Following an introduction to the OS/2 and PM environment and how it works, the book offers detailed coverage of OS/2 2.0 programming in COBOL, including:

- COBOL coding references for the most common PM calls
- sample COBOL code that forms the nucleus of every COBOL-PM program which can be “cut and pasted” into your actual programs
- and much more!

The book’s methodical format groups PM calls as they relate to building a part of a window or dialog, not by an artificial PM collection, such as Gpi calls. For example, you will find a discussion of Resource Script files included in the chapter about adding menus to the frame window, because Resource Script files are required when working with menus. This step-by-step progression through PM complexities will help readers gain a better understanding of how all the pieces fit together.

For companies and programmers who have used COBOL on large and midrange systems, the ability to use COBOL as the primary programming language will greatly alter the cost equation for converting to the OS/2 and the Presentation Manager. With *The COBOL Presentation Manager Programming Guide*, you will be able to effectively begin the transition to the OS/2 Environment without rewriting stable, well-performing applications for the sake of a new interface or a new platform.

About the Author

David M. Dill worked for IBM for over 23 years, most recently in the area of customer support, specializing in installations and usage support for PCs. For the past four years, he provided technical support for the OS/2 Standard Edition operating system, including the OS/2 Kernel, Presentation Manager, the OS/2 printing subsystem, the programming Toolkit, and COBOL programming under OS/2 and Presentation Manager.

For OS/2 versions 1.3 and 2.0

*Cover design: S R Plourde/Howling Moon Graphics
OS/2 is a registered trade mark of the IBM Corporation.*

G362-0010-00

